Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **MAY 2012**	2. REPORT TYPE	3. DATES COVERED **00-00-2012 to 00-00-2012**

4. TITLE AND SUBTITLE **Improving Systematic Constraint-driven Analysis Using Incremental and Parallel Techniques**	5a. CONTRACT NUMBER 5b. GRANT NUMBER 5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER 5e. TASK NUMBER 5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **The University of Texas at Austin, Austin,TX, 78712**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S) 11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

1/20

14. ABSTRACT

This dissertation introduces Pikse, a novel methodology for more effective and efficient checking of code conformance to specifications using parallel and incremental techniques, describes a prototype implementation that embodies the methodology, and presents experiments that demonstrate its efficacy. Pikse has at its foundation a well-studied approach ? systematic constraint-driven analysis ? that has two common forms: (1) constraint-based testing ? where logical constraints that define desired inputs and expected program behavior are used for test input generation and correctness checking, say to perform black-box testing; and (2) symbolic execution ? where a systematic exploration of (bounded) program paths using symbolic input values is used to check properties of program behavior say to perform white-box testing.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**216**	

The Dissertation Committee for Junaid Haroon Siddiqui
certifies that this is the approved version of the following dissertation:

Improving Systematic Constraint-driven Analysis using Incremental and Parallel Techniques

Committee:

____
Sarfraz Khurshid, Supervisor

____
Dewayne Perry

____
Adnan Aziz

____
Derek Chiou

____
Darko Marinov

Improving Systematic Constraint-driven Analysis using Incremental and Parallel Techniques

by

Junaid Haroon Siddiqui, B.S.; M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2012

Acknowledgments

This thesis was made possible by contributions from a large number of people. Foremost, I would like to thank my adviser Sarfraz Khurshid. Sarfraz has been a mentor throughout my stay in Austin and gave me good advise on everything from choosing an apartment to choosing the right courses. Whenever I had a crudely crafted idea and I conveyed to him in the most rudimentary way, he would make some sense out of it and help me formulate it into a concrete technique and how to evaluate it. And when the evaluation was done, he always pushed to write a paper and submit it very soon. A couple of times, we worked till 3 or 4AM on a paper submission deadline. Those were memorable times.

Dewayne Perry, Derek Chiou, Adnan Aziz, and Darko Marinov agreed to serve on my PhD committee and provided me with very insightful comments in my proposal defense that helped me shape my research in the last couple of years. In particular, I would like to thank Darko for periodically providing extensive comments and discussions on various topics in this thesis. Derek gave me a lot of ideas in the domain of parallel algorithms during the “Parallel Architecture” I took from him. That course has a lot of influence in the parallel and work stealing approaches in this thesis. Courses I studied from David Zuckerman, Craig Chase, and Scott Nettles also provided useful fundamentals that helped me shape my research focus.

I would also like to thank the anonymous reviewers at ICST 2009, ICFEM 2009, ASE 2009, ICSTE 2010, ICFEM 2011, SAC 2012, and ICST 2012 for reviewing my work and providing me with useful comments.

When a series of coincidences resulted in none of our group members being

able to make it to ICFEM 2011, Shaz Qadeer of Microsoft Research accepted our request to present our work at ICFEM 2011. I would like to thank Shaz for helping us out.

I would also like to thank Anna Gringuaze and Andrew Kadatch for mentoring me during my two internships. Anna mentored me during my internship at Microsoft Research. I worked on making a dynamic analysis algorithm incremental. It was a great learning experience. Andrew mentored me during my internship at Google Kirkland. I worked on analyzing and modeling latency of a cloud based subsystem.

Members of my research group provided useful comments and ideas on my work in group meetings and individual discussions. In particular, former group members Danua Shao, Eddy Reyes, and Bassem Elkarablieh provided detailed comments on some of the work presented here.

This material is based upon work partially supported by the Fulbright Program, the National Science Foundation under Grant Nos. IIS-0438967, CNS-0615372, CCF-0702680, CCF-0746856, and CCF-0845628, and AFOSR grant FA9550-09-1-0351, and the Naval Undersea Warfare Center. Texas Advanced Computing Center (TACC) provided the resources for experimenting with the parallel algorithms proposed in this thesis.

I also want to thank Belal Hashmi, who mentored me during my stay at NUCES university, and encouraged me to get higher education and pursue an academic career. Although, the only thing he didn't want me do was a PhD in Software Engineering but interestingly that's exactly what I am doing. I was also influenced in my decision to pursue a PhD by my close friends Atif Mehdi, Yasir Khan, and Umar Suleman, all three are currently pursuing PhD in Robotics at different universities.

My decision to join UT in particular was most influenced by Zubair Malik. He joined UT an year before me and encouraged me to apply here and join UT. Since then, we have always shared research ideas and have heated debates about them.

My friends at UT Umar Farooq, Owais Khan, Khubaib, Rashid Kaleem, Amber Hassaan, and Faisal Iqbal gave me a great time in Austin. Amber and Faisal hosted me when I first came here. I rented my first apartment with Umar and we are roommates since then.

Most importantly, I would like to thank my wife Huma Qureshi for supporting me throughout my PhD. She works at Microsoft which means that we are often apart by about two thousand miles. Yet, her love, support, and encouragement is always with me. In fact, she has always been a source of motivation for me since I met her nine years ago. The completion of this PhD owes the most to her.

In the end, I would like to thank my father, mother, and siblings, who have all been very supportive during my PhD. Even when I forget to call for days, I always hear the same energetic voices that encourage me to push a little more and complete my work. I look forward to see them all.

Improving Systematic Constraint-driven Analysis using Incremental and Parallel Techniques

Publication No. ___________________

Junaid Haroon Siddiqui, Ph.D.
The University of Texas at Austin, 2012

Supervisor: Sarfraz Khurshid

This dissertation introduces Pikse, a novel methodology for more effective and efficient checking of code conformance to specifications using parallel and incremental techniques, describes a prototype implementation that embodies the methodology, and presents experiments that demonstrate its efficacy. Pikse has at its foundation a well-studied approach – *systematic constraint-driven analysis* – that has two common forms: (1) *constraint-based testing* – where logical constraints that define desired inputs and expected program behavior are used for test input generation and correctness checking, say to perform *black-box testing*; and (2) *symbolic execution* – where a systematic exploration of (bounded) program paths using *symbolic* input values is used to check properties of program behavior, say to perform *white-box testing*.

Our insight at the heart of Pikse is that for certain path-based analyses, (1) the state of a run of the analysis can be encoded compactly, which provides a basis for parallel techniques that have low communication overhead; and (2) iterations performed by the analysis have commonalities, which provides the basis for incremental techniques that re-use results of computations common to successive iterations.

We embody our insight into a suite of parallel and incremental techniques that enable more effective and efficient constraint-driven analysis. Moreover, our techniques work in tandem, for example, for combined black-box constraint-based input generation with white-box symbolic execution. We present a series of experiments to evaluate our techniques. Experimental results show Pikse enables significant speedups over previous state-of-the-art.

Contents

List of Tables

List of Figures

List of Listings

List of Algorithms

Chapter 1

Introduction

Software failures are expensive. The most commonly used methodology for validating quality of software is testing. Traditional approaches for testing are often ad hoc, manual, and even ineffective. Researchers have developed various approaches to automate testing [29, 46, 52, 53, 65, 66]. The last decade [10, 42, 63, 64, 91] has seen substantial improvements in automated testing approaches, which are now capable of handling programs that use advanced constructs of modern programming languages. Moreover, automation has enabled *systematic* testing, also known as *bounded exhaustive testing*, where a program is checked against all inputs (with desired properties) up to a given size [63]. Systematic testing takes its inspiration in part from model checking [55] – a methodology that is highly successful for hardware verification – where exhaustive exploration of a bounded state space is a fundamental idea. A number of studies have shown the ability of systematic testing to find bugs in a variety of programs, including those that perform intricate operations on structurally complex input data and that it holds much promise in improving reliability of software systems [17].

1.1 Problem context and description: Systematic constraint-driven analysis

Constraint-based testing [22, 24, 53, 65, 87] is a well-studied technique for systematic testing, where user-provided logical constraints – *input constraints* that define properties of desired inputs and *output constraints* that define properties of expected program behavior – are solved using automatic constraint solvers to generate test inputs and check correctness. To illustrate, consider testing a method in an object-oriented program; the method's *pre-condition* can be used as the input constraint to generate valid pre-states to invoke the method, and it's *post-condition* can be used as the output constraint to check that the expected relation between pre-states and post-states holds for the method executions tested.

Typically, constraint-based testing is used in a *black-box* setting where the code under test is simply run against desired inputs to generate outputs, which are checked. However, more general *constraint-driven analysis* can also be used in a *white-box* setting where the language constructs that implement the code, e.g, statements and branches, are leveraged, e.g., to direct input generation to maximize branch coverage. A particularly popular technique for white-box testing is *symbolic execution* [22, 65] – a program analysis, which is also constraint-driven.

1.1.1 Black-box testing

Writing constraints. Logical constraints are *declarative* in nature, i.e., they describe *what* (and not *how*). Thus, it would seem most natural to write them using a declarative language [55]. However, for programmers who write code only in commonly used imperative languages, such as C/C++, writing constraints using a likely unfamiliar declarative paradigm may pose a substantial learning burden and diminish the attractiveness of constraint-driven analysis. A common approach to facilitate writing constraints is to support *imperative constraints*, e.g., predicate methods that return true if their inputs satisfy the constraints they represent and false otherwise. Thus, imperative constraints allow programmers to formulate desired properties using constructs of the same language they use for programming.

Solving constraints. The key enabling technology in systematic constraint-driven analysis is automatic constraint solving. Recent years have seen notable advances in this technology [79, 101]. Moreover, these advances have favorably been backed by the growth in raw computation power. As a result, constraint solvers today are able to efficiently solve several useful classes of formulas and their application to software verification is rapidly growing. For imperative constraints written as predicates, a novel solving technique – *execution-driven pruning* – was introduced by the Korat framework [10], where the basic idea is to use repeated predicate executions on candidate inputs to observe properties that make inputs invalid and prune inputs with those properties from the space of all inputs. Figure 1.1 gives an example C/C++ predicate and the inputs generated by Korat using a bound of 3 nodes.

1.1.2 White-box testing

Constraints based on program paths. Symbolic execution [22, 65] is a program analysis technique that allows constraints to be derived from the code. Specifically, the technique explores all program paths (up to a given bound on path length) and for each path, builds a *path condition*, which represents the branching conditions along the path – a solution to a feasible path condition is a program input that executes that path. Thus, a common application of symbolic execution is input generation for (bounded) path coverage in white-box testing. Figure 1.2 illustrates symbolic execution on an example C/C++ program to generate tests for path coverage.

Figure 1.1 Korat produces all structures of a given size using a predicate function.

```
struct BinaryTree {
  struct Node {
    Node* left;
    Node* right;
  };
  Node* root;
  int size;
  bool repOk() {
    std::set<Node*> v; // visited
    std::stack<Node*> worklist;
    if( root ) {
      worklist.push( root );
      v.insert( root );
    }
    while( !worklist.empty() ) {
      Node* current = worklist.top();
      worklist.pop();
      if( current->left ) {
        if(!v.insert(current->left).second)
          return false;
        worklist.push( current->left );
      }
      if( current->right ) {
        if(!v.insert(current->right).second)
          return false;
        worklist.push( current->right );
      }
    }
    return visited.size() == size;
  }
};
```

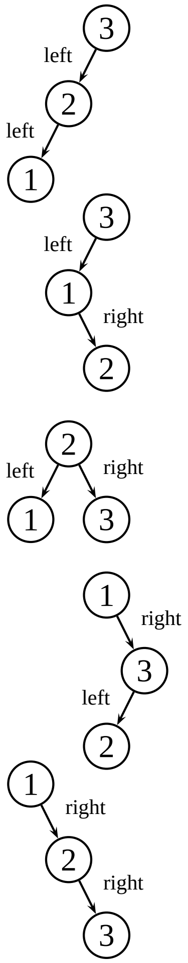
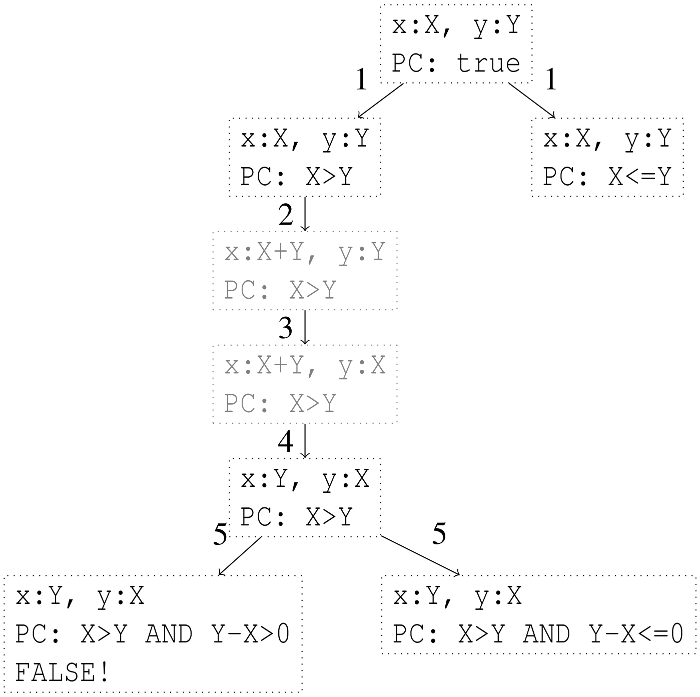


Figure 1.2 Code that swaps two integers and the corresponding symbolic execution tree, where transitions are labeled with program control points [64]

```
1: int x, y;
2: if (x > y) {
3:   x = x + y;
4:   y = x - y;
5:   x = x - y;
6:   if (x - y > 0)
7:     assert(false);
8: }
```



1.1.3 Correctness properties

The focus of this dissertation is systematic constraint-driven analysis for checking functional correctness properties of data-driven applications, e.g., programs that operate on dynamically allocated data and perform destructive updates on the memory heap. These functional properties include absence of run-time failures, such as segmentation faults, as well as richer properties that are explicitly provided by the user as test assertions. While checking of other forms of properties, e.g., properties of control, performance, and space efficiency, may also be possible using constraints, we do not consider them here.

1.2 Our solution

This dissertation introduces Pikse, a novel methodology for more effective and efficient checking of code conformance to specifications using parallel and incremental techniques. Specifically, Pikse enhances systematic constraint-driven analysis by (1) introducing parallel and incremental algorithms to optimize solving of imperative constraints and enabling more efficient black-box testing; (2) introducing parallel and incremental algorithms to optimize symbolic execution and enabling more efficient white-box testing; and (3) introducing *staged* analysis for constraint-based testing to enable more efficient and effective combined black-box and white-box testing.

Our insight into Pikse's parallel and incremental techniques is that for certain constraint-driven analyses, such as imperative constraint solving and symbolic execution: (1) the state of an analysis run can be encoded compactly, which provides a basis for parallel analysis that has low communication overhead and allows effective load balancing; and (2) the analysis performs an iterative computation that has significant commonalities in successive iterations, which provides the basis for incremental analysis that re-uses results of computations common to successive iterations. Moreover, our insight into Pikse's staged analysis is that an input constraint can be solved *partially* to build *abstract* test suites that can be concretized *on demand* using symbolic execution using each abstract test, which provides the basis for re-using abstract suites across different programs, and reducing the size of suites and hence the number of tests to run.

1.3 Contributions

1.3.1 Symbolic Execution

To enhance symbolic execution for white-box testing, we make the following contributions.

- **Test input as analysis state.** We introduce the idea of *encoding* the state of symbolic execution using a single test input using a given *branch exploration strategy*.

- **Ranged symbolic execution.** Using two test inputs, ranged symbolic execution defines a range of paths to be analyzed under a given branch exploration strategy. Thus, a symbolic execution problem of systematically analyzing bounded execution paths is divided into a number of sub-problems using simply a set of test inputs.

- **Dynamic range refinement using work stealing.** We introduce load-balancing for parallel symbolic execution using dynamically defined ranges that are refined using work stealing.

- **Parallel Symbolic Execution using master/slave architecture:** Additionally we present a different technique based on a novel parallel algorithm for executing symbolic execution on a parallel cluster. We also compare techniques for stopping distribution of work when small enough problems are formed for reducing communication overhead while load balancing as well.

1.3.2 Constraint-based testing

To enhance the solving of imperative constraints for improved black-box testing, we make the following contributions.

- **Lightweight static data-flow analysis for constraint solving:** We introduce the idea of utilizing def-use analysis in optimizing repeated predicate executions on similar inputs.
- **Multi-value comparisons.** We introduce the idea of comparing sets of values with a desired value to compute the predicate's output on its future executions that are thus forwarded.
- **Focused Generation:** We propose a way to allow some structural constraints to be solved exhaustively but others to be solved for a single solution. For instance, we can test a structure that contains instances of pre-tested structures, by exploring the outer structure exhaustively but not the inner structure.
- **Parallel Algorithm for Test Generation:** We present PKorat, a scalable parallel algorithm based on Korat for generation of structurally complex test inputs.

1.3.3 Combined black-box white-box testing

To combine black-box and white-box testing for efficient analysis, we make the following contributions.

- **Abstract symbolic tests:** We introduce abstract symbolic tests that are a combination of concrete tests and constrained symbolic elements. These tests are not executable using conventional execution, rather they provide a basis for symbolic execution of programs with pre-conditions.

- **Staged symbolic execution:** We introduce the technique of performing symbolic execution in stages, where the first stage generates abstract symbolic tests, which the second stage uses for systematic testing — each abstract symbolic test is dynamically expanded into a number of concrete tests depending on the control-flow complexity of the program under test. Staged symbolic execution allows both a reduction in test suite size without a loss in its quality, as well as a novel re-use of tests.

- **Symbolic execution for declarative programs:** We introduce the idea of symbolic execution for declarative programs written in analyzable notations, similar to symbolic execution of imperative programs. We enable the possibilities of using this as the first stage in staged testing.

- **Symbolic execution for Alloy models:** We present our approach for symbolic execution of Alloy, and provide an extensible technique to support various symbolic types and operators.

1.3.4 Implementation and experiments

In implementing and evaluating the above algorithms, we make the following contributions.

- **Implementation:** We provide an efficient implementation of our algorithms. We build our techniques on the Korat tool for constraint-based testing [10], KLEE [14] – an open-source symbolic execution implementation, and the CREST [12] symbolic execution tool. We employ MPI [100] for our parallel implementations and run them on the Texas Advanced Computing Center (TACC)[1].

- **Experiments:** We perform a series of experiments to validate the algorithms and techniques presented in this thesis. Our subjects include GREP – a C program with 15K lines of code, GNU Coreutils – a suite of widely deployed Unix utilities, Java program generation [27], and complex structures like red-black trees and dynamic order statistics [26].

- **Empirical Study:** We perform an extensive study of existing techniques for constraint solving. We conduct a controlled experiment for performance analysis of different constraint solving techniques. We attempt to quantify the trade-offs of these techniques in writing constraints and in processing their outputs.

1.4 Organization

This thesis document is organized as follows:

[1] `http://tacc.utexas.edu`

Chapter 2 gives a background on symbolic execution and constraint-based testing and explains their working with examples.

Chapter 3 discusses how our incremental and parallel techniques scale symbolic execution in novel ways. We give a series of experiments showing the benefits.

Chapter 4 discusses the improvements in constraint-based testing which is often used for black-box testing.

Chapter 5 introduces “staged testing” where dynamic analysis can be divided in different stages and each stage can benefit from a selection of different techniques.

Chapter 6 presents an empirical study of four techniques comparing them for solving structural constraints. It also introduces the idea of symbolic execution in Alloy.

Chapter 7 discusses related work in the domains of symbolic execution, SAT solver based techniques, constraint-based testing, and parallel algorithms for software checking.

Chapter 2

Background

In this chapter, we present the basics of symbolic execution, constraint-based testing using Korat, and the Alloy tool based on SAT solving. We explain the key ideas using illustrative examples.

2.1 Symbolic Execution

Symbolic execution [22, 65] is a technique first presented over three decades ago for systematic exploration of behaviors of imperative programs using symbolic inputs, which characterize classes of concrete inputs. The key idea behind symbolic execution is to explore (feasible) execution paths by building *path conditions* that define properties required of inputs to execute the corresponding paths. The rich structure of path conditions enables a variety of powerful static and dynamic analyses. However, traditional applications of symbolic execution have largely been limited to small illustrative examples, since utilizing path conditions in automated analysis requires much computation power, particularly for non-trivial programs that have long execution paths with complex control flow. During recent years, many advances has been made in constraint solving technology [79] and additionally, raw computation power has increased substantially. These advancements have led to a resurgence of symbolic execution, and new variants that perform *partial* symbolic execution have become particularly popular for systematic bug finding [17] in programs written in commonly used languages such as C/C++, C#, and Java.

Traditional symbolic execution is a combination of static analysis and theorem proving. In symbolic execution, operations are performed on symbolic variables instead of actual data. On branches, symbolic execution is forked with opposite constraints on symbolic variables in each forked branch. At times, the constraints on symbolic variables can become unsatisfiable signaling unreachable code. Otherwise, end of the function is reached and a formula on symbolic variables is formed. A solution to this formula will give a set of values that will direct an actual execution along the same path.

2.1.1 Example

Forward symbolic execution is a technique for executing a program on symbolic values [65]. There are two fundamental aspects of symbolic execution: (1) defining semantics of operations that are originally defined for concrete values and (2) maintaining a *path condition* for the current program path being executed – a path condition specifies necessary constraints on input variables that must be satisfied to execute the corresponding path.

As an example, consider the following program that returns the absolute value of its input:

```
       static int abs(int x) {
L1.      int result;
L2.      if (x < 0)
L3.        result = -x;
L4.      else result = x;
L5.      return result;      }
```

To symbolically execute this program, we consider its behavior on a primitive integer input, say X. We make no assumptions about the value of X (except what can be deduced from the type declaration). So, when we encounter a conditional statement, we consider both possible outcomes of the condition. To perform operations on symbols, we treat them simply as variables, e.g., the statement on L3 updates the value of result to be -X. Of course, a tool for symbolic execution needs to modify the type of result to note updates involving symbols and to provide support for manipulating expressions, such as -X.

Symbolic execution of the above program explores the following two paths:

```
path 1:     [X < 0] L1 -> L2 -> L3 -> L5
path 2:     [X >= 0] L1 -> L2 -> L4 -> L5
```

Note that for each path that is explored, there is a corresponding path condition (shown in square brackets). While execution on a concrete input would have followed exactly one of these two paths, symbolic execution explores both.

2.2 Constraint-based testing with Korat

The Korat framework [10] introduced a novel technique for solving imperative constraints – *execution-driven solving* – where a bounded space of candidate inputs is systematically explored by executing the given predicate on candidate inputs and monitoring the executions to filter and prune invalid candidates from the input space. This technique has been used effectively for finding bugs in a number of applications [38, 76, 105, 106] and similar techniques lay at the basis of other effective frameworks for systematic bug finding, e.g., lazy initialization in generalized symbolic execution [64] and the UC-KLEE framework [88].

Korat implements a backtracking search and uses the predicate's executions to prune large portions of its input space. Korat runs the predicate on a candidate input, monitors the fields accessed by the predicate, backtracks on the last field accessed to generate a new candidate by making a non-deterministic assignment to that field, and re-executes the predicate. Korat's backtracking, driven by field-access monitoring, provides significant pruning of input spaces, which are very large even for small sized inputs.

Korat uses an imperative predicate as input. An imperative predicate is a predicate function written in an imperative language, as opposed to declarative language, to check the structural properties of a complex structure. It is conventionally called `repOk`. In the object-oriented paradigm, such a function is called a *class invariant* [73].

Algorithm 2.1 describes the Korat algorithm as a recursive function. This recursive function starts with the candidate vector with all zeroes. It builds a C++ object structure using BUILDCANDIDATE. BUILDCANDIDATE makes field assignments according to candidate vector indices and generates an actual candidate from a candidate vector. It then tests the candidate using `repOk`. Then for all accessed fields up to a field pointing to a non-zero index, it recursively tests candidates for all non-zero indices of that field up to the maximum index given by NONISOMAX.

Further pruning is done in Korat to avoid isomorphic structures. Isomorphic structures are structures that only differ in object identities. Programs are not usually concerned with the identity of an object, e.g. the actual memory address in C++ or the object hash code in Java. The identities of n objects can be interchanged in $n!$ ways. Isomorphic structure avoidance therefore prunes a large portion of search space.

Algorithm 2.1: The Korat algorithm written as a recursive function. It builds a C++ object structure using BuildCandidate and tests it using `repOk`. It recursively tests candidates for all non-zero indices of the last accessed field up to the maximum index given by NonIsoMax.

```
input  : candidateV
candidate ← BuildCandidate(candidateV);
(predicate, accessFields) ← repOk(candidate);
if predicate then
    ValidCandidate(candidate);
end
while Size(accessFields)>0 ∧ candidateV[Top(accessFields)]=0 do
    for i ← 1, NonIsoMax(candidateV, accessFields) do
        candidateV[Top(accessFields)] ← i;
        Korat(candidateV);
    end
    candidateV[Top(accessFields)] ← 0;
    Pop(accessFields);
end
```

Algorithm 2.2: The NonIsoMax algorithm used by Korat to ensure that isomorphic structures are not produced.

```
input  : candidateV, accessFields
f ← Top(accessFields);
if Primitive(f) then
    return MaxDomainIndex(f) ;
else
    t ← 0;;
    forall the i ∈ accessFields do
        if SameDomain(i, f) ∧ t<candidateV[i] then
            t ← candidateV[i];
        end
    end
    return Min(t+1,MaxDomainIndex(f)) ;
end
```

2.2.1 Example: Binary search tree

The binary search tree class (BST) is defined in Listing 2.1. It contains a sub-class `Node` that represents a single node in the BST. The `Node` contains `left` and `right` pointers to other nodes, a `parent` pointer and an integer `data` field. The outer class contains a pointer to the `root` node and the number of nodes in the tree in the `size` field. Two methods must be provided for constraint solving: a `finitize` method that describes bounds for analysis (called *finitization*) and a `repOk` predicate method that tells if a particular instance of BST is *valid* or not (also called the *class invariant*).

Listing 2.1. Binary search tree class definition.

```
class BST {
  struct Node {
    Node* left;
    Node* right;
    Node* parent;
    int data;
  };
  Node* root;
  int size;
 public:
  static Finitization* finitize(int size);
  bool repOk();
  void add(int data) {
    // method under test
  }
};
```

We provide the finitization of BST in Listing 2.2. We create a domain of `Node` objects and require `root`, `left`, `right`, and `parent` to take values only from this domain. For `data` and `size` fields, we create integer domains of appropriate sizes.

Listing 2.2. Finitization for Korat.

```
Finitization* BST::finitize(int c) {
  Finitization* f = Finitization::create<BST>();
  Domain<Node>* nodes = f->domain<Node>(c);
  f->set(&BST::root, nodes);
  f->set(&BST::size, f->domain<int>(c, c));
  f->set(&Node::left, nodes);
  f->set(&Node::right, nodes);
  f->set(&Node::parent, nodes);
  f->set(&Node::data, f->domain<int>(1, c));
  return f;
}
```

Next, we provide the repOk function for BST in Listing 2.3. It is the class invariant and checks the BST properties. These properties are: (1) acyclicity along left and right pointers, (2) correct parent pointers, and (3) larger data values are stored in right sub-tree while smaller data values are stored in left sub-tree. The given function checks these properties using a work-list based algorithm.

Korat takes as input a class definition with its class invariant and a finitization function (provided by repOk and finitize methods in the above example).

To start solving the constraint, the Korat algorithm forms an initial candidate structure to test. The candidate is formed by assigning every field the first value from its domain specified in the finitization. Korat executes repOk on this candidate to check if it is valid.

During the repOk execution, Korat monitors field accesses and builds a field-access list. After repOk finishes, Korat picks a new value for the last accessed field from its field domain and runs repOk again. If there are no more values in its field domain, it backtracks to the field accessed before it. This way Korat explores the state-space without testing every possible combination of values in the

Listing 2.3. Class invariant for binary search tree.

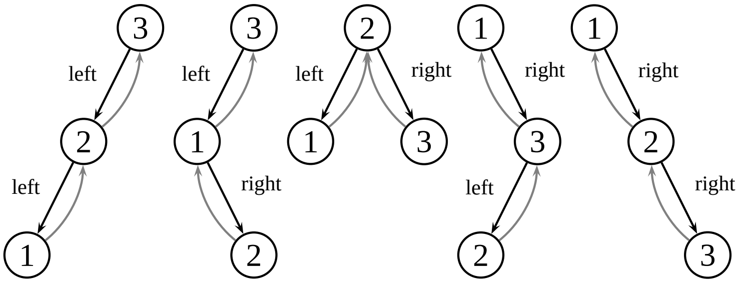
```
bool BST::repOk() {
  set<Node*> visited;
  stack<tuple<Node*, int, int> > wl;
  if (root) {
    wl.push(make_tuple(root,
    numeric_limits<int>::min(),
    numeric_limits<int>::max()));
    visited.insert(root);
  }
  while (!wl.empty()) {
    Node* c = get<0>(wl.top());
    int min = get<1>(wl.top());
    int max = get<2>(wl.top());
    wl.pop();
    if (c->data < min || c->data > max)
      return false;
    if (c->left) {
      if (!visited.insert(c->left).second)
        return false;
      if (c->left->parent != c)
        return false;
      wl.push(make_tuple(c->left, min, c->data-1));
    }
    if (c->right) {
      if (!visited.insert(c->right).second)
        return false;
      if (c->right->parent != c)
        return false;
      wl.push(make_tuple(c->right, c->data+1, max));
    }
  }
  return size == visited.size();
}
```

field domains.

Korat produces non-isomorphic inputs. Non-isomorphic inputs differ only in the identity of objects used and provide no additional fault-finding ability in testing code. To produce non-isomorphic inputs, Korat records the values in the field domain of reference fields that are accessed by other reference fields of the same type. It only backtracks to `null`, values also referenced by other fields of the same type, and one new value. For example, if N_0 and N_1 are used by `root` and `left` pointers respectively, the `right` pointer will take a value from $\{$`null`$, N_0, N_1, N_2\}$. Choosing another value N_3 would form a structure isomorphic to the one formed by using N_2.

Given the class definition, finitization, and class invariant, our desired output is a set of all concrete structures of a given size. For size 3 these structures are shown in Figure 2.1. We expect the constraint solver to list all of the five structures and only these five structures.

Figure 2.1 Binary search trees of 3 nodes with parent pointers.



Korat starts its search from an empty tree with `root=null` and backtracks on accessed fields to try other values. To explain the working of Korat we describe its progression between two valid candidates shown at the right and left extremes of Figure 4.1 on page 74.

When Korat analyzes Figure 4.1(a) using repOk, the fields accessed are (root, N_0.data, N_0.left, N_0.right, N_1.parent, N_1.data, N_1.left, N_1.right, N_2.parent, N_2.data, N_2.left, N_2.right, size). After marking this as a valid candidate, it backtracks to N_2.right as size is already at its maximum value. All choices for N_2.right result in cyclic structures, so Korat backtracks to N_2.left which also results in cyclic structures. After each of these executions, Korat uses the "field access list" generated as a result of the last execution. Since we assume that the repOk function is deterministic, the initial part of the field access list is the same.

After backtracking past other fields, Korat resets N_1.right to null and backtracks to try other values for N_1.left. When it tries N_1.left=N_2, it fails because N_2.parent=null (its initial value). However the field access list has a new field and Korat tries its other values. N_2.parent=N_1 works but repOk fails because N_2.data$\not>$ 1. This is Figure 4.1(b). In total, Korat performs 16 repOk executions between these two states. Korat proceeds this way until all values are exhausted and finds all valid structures within the given bounds.

2.3 Alloy

Alloy is a first-order declarative logic based on sets and relations, and is supported by its fully automatic, SAT-based analyzer [55]. The Alloy tool-set is becoming popular for academic research and teaching as well as for designing dependable software in industry. The powerful analysis performed by the analyzer makes Alloy particularly attractive for modeling and checking a variety of systems, including those with complex structural constraints – SAT provides a particularly efficient analysis engine for such constraints.

An Alloy specification is a sequence of paragraphs that either introduce new types or record constraints on *fields* of existing types. Alloy assumes a universe of atoms partitioned into subsets, each of which is associated with a basic type. Details of the Alloy notation and of the Alloy Analyzer can be found in [55].

2.3.1 Example

Acyclic lists can be modeled in Alloy with the following specification

```
one sig AcylicList {
  header: lone Node,
  size: Int }
sig Node {
  data: Int,
  nextNode: lone Node }
pred Acyclic(l: AcylicList) {
  all n: l·header·*nextNode | n !in n·^nextNode }
```

The *signature* declarations `AcylicList` and `Node` introduce two uninterpreted types, along with functions `header : AcylicList → Node`, `size : AcylicList → Int`, `data : Node → Int`, and `nextNode : Node → Node`. `header` and `nextNode` are partial functions, indicated by the declaration `lone`.

The Alloy *predicate* Acyclic, when invoked, constrains its input `l` to be acyclic. The dot operator '`.`' represents relational image, '`~`' represents transpose, '`^`' represents transitive closure, and '`*`' denotes reflexive transitive closure.

The quantifier `all` stands for universal quantification. For instance, the constraint `all n: l.header.*nextNode | F` holds if and only if evaluation of the *formula* `F` holds for each atom in the transitive closure of `nextNode` starting from `l.header`. The quantifier `lone` stands for "at most one". There are also quantifiers `some` and `no` for existential and universal quantification.

Given an Alloy specification, the Alloy Analyzer automatically finds *instances* that satisfy the specification, i.e., the valuations of relations and signatures that make all the applicable constraints in the specification true. Alloy Analyzer finds instances within a pre-specified *scope* – bound on the universe of discourse. Alloy Analyzer can also enumerate non-isomorphic instances.

Chapter 3

Constraint-driven analysis for white-box testing

This chapter describes the Pikse suite of techniques for improving constraint-driven analysis for white-box testing. Specifically, we describe ranged symbolic execution (Section 3.1), which enables parallel and incremental symbolic execution, and ParSym (Section 3.2), which presents a different technique for parallel symbolic execution. ParSym was initially presented at ICSTE 2010 [94].

3.1 Ranged symbolic execution

A key limiting factor of symbolic execution remains its inherently complex path-based analysis. Several recent research projects have attempted to address this basic limitation by devising novel techniques, including compositional [40], incremental [86], and parallel [16, 43, 94, 103] techniques. While each of these techniques offers its benefits (Chapter 7), a basic property of existing techniques is the need to apply them to completion in a single execution if *completeness* of analysis (i.e., complete exploration of the bounded space of paths) is desired. Thus, for example, if a technique times out, we must re-apply it for a greater time bound, which can represent a costly waste of computations that were performed before the time out.

In this section, we present *ranged symbolic execution* (under submission), a novel technique for scaling symbolic execution. Our key insight is that the state of a symbolic execution run can, rather surprisingly, be encoded succinctly by a test input – specifically, by the input that executes the last terminating (feasible) path explored by symbolic execution. By defining a fixed *branch exploration ordering* – e.g., taking the false branch before taking the true branch at each non-deterministic branch point during the exploration – an operation already fixed by common implementations of symbolic execution [2, 14], we have that each test input partitions the space of (bounded) paths under symbolic execution into two sets: *explored* paths and *unexplored* paths. Moreover, the branch exploration ordering defines a *linear order* among test inputs; specifically, for any two inputs (that do not execute the same path or lead to an infinite loop), the branching structure of the corresponding paths defines which of the two paths will be explored first by symbolic execution. Thus, an ordered pair of tests, say $\langle\tau,\tau'\rangle$, defines a *range* of (bounded) paths $[\rho_1,\ldots,\rho_k]$ where path ρ_1 is executed by τ and path ρ_k is executed by τ', and for $1 \leq i < k$, path ρ_{i+1} is explored immediately after path ρ_i.

Encoding the symbolic execution state as a test input enables a variety of novel ways to distribute the exploration in symbolic execution to scale it, both in a sequential setting and in a parallel setting. The encoding allows dividing the problem of symbolic execution into several sub-problems of ranged symbolic execution, which have minimal overlap and can be solved separately. It also allows effective load balancing in a parallel setting using *work stealing* with minimal overhead due to the compactness of a test input.

3.1.1 Illustrative Example

Forward symbolic execution is a technique for executing a program on symbolic values [22, 65]. There are two fundamental aspects of symbolic execution: (1) defining semantics of operations that are originally defined for concrete values and

(2) maintaining a *path condition* for the current program path being executed – a path condition specifies necessary constraints on input variables that must be satisfied to execute the corresponding path.

As an example, consider the following program that returns the middle of three integer values.

```
1:  int mid(int x, int y, int z) {
2:    if (x<y) {
3:      if (y<z) return y;
4:      else if (x<z) return z;
5:      else return x;
6:    } else if (x<z) return x;
7:    else if (y<z) return z;
8:    else return y; }
```

To symbolically execute this program, we consider its behavior on primitive integers input, say X, Y, and Z. We make no assumptions about the value of these variables (except what can be deduced from the type declaration). So, when we encounter a conditional statement, we consider both possible outcomes of the condition. To perform operations on symbols, we treat them simply as variables and calculate the expression.

Symbolic execution of the above program explores the following six paths:

```
path 1:  [X < Y < Z] L2:  -> L3:
path 2:  [X < Z < Y] L2:  -> L3:  -> L4:
path 3:  [Z < X < Y] L2:  -> L3:  -> L4:  -> L5:
path 4:  [Y < X < Z] L2:  -> L6:
path 5:  [Y < Z < X] L2:  -> L6:  -> L7:
path 6:  [Z < Y < X] L2:  -> L6:  -> L7:  -> L8:
```

Note that for each path that is explored, there is a corresponding path condition (shown in square brackets). While execution on a concrete input would have followed exactly one of these paths, symbolic execution explores all six paths.

The path conditions for each of these paths can be solved using off-the-shelf SAT solvers for concrete tests that exercise the particular path. For example, path 2 can be solved to `X=1`, `Y=3`, and `Z=2`.

Ranged symbolic execution enables symbolic exploration between two given paths. For example, if path 2 and path 4 are given, it can explore paths 2, 3, and 4. In fact, it only needs the concrete solution that satisfies the corresponding path condition. Therefore it is efficient to store and pass paths. Ranged symbolic execution builds on a number of key observations we make:

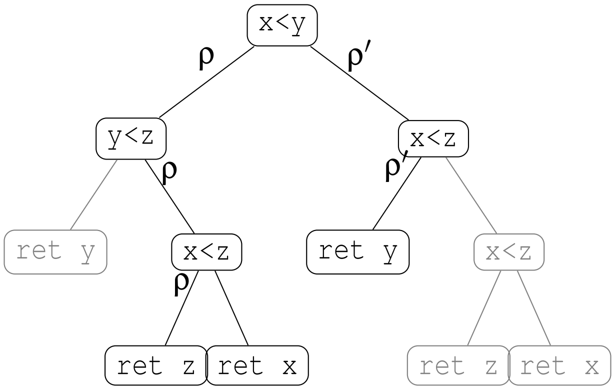
- A concrete solution corresponds to exactly one path in code and it can be used to re-build the path condition that leads to that path. Solving a path condition to find concrete inputs is computationally intensive. However, checking if a given solution satisfies a set of constraints is very light-weight. Thus we can symbolically execute the method again and at every branch only choose the direction satisfied by the concrete test.

- We can define an ordering on all paths if the true side of every branch is always considered before the false side. Since, every concrete test can be converted to a path, the ordering can be defined over any set of concrete inputs.

- Using two concrete inputs, we can find two paths in the program and we can restrict symbolic execution between these paths according to the ordering defined above. We call this *ranged symbolic execution*.

For example, consider that we are given test inputs τ(`X=1, Y=3, Z=2`) and τ'(`X=2, Y=1, Z=3`) which take paths ρ and ρ' in code, and we want to symbolically execute the range between them. We show this example in Figure 3.1. We start symbolic execution as normal and at the first comparison `x<y`, we note that ρ traverses the `true` branch while ρ' traverses the `false` branch. At this point, we also know that $\rho < \rho'$ in the ordering we defined. Thus, when $x < y$, we only explore what comes after ρ in the ordering and when $x \not< y$ we explore what comes before ρ'. At the next comparison `y<z` we skip the `true` branch and only explore the `false` branch satisfied by ρ. Similarly we can skip three states using ρ'. Skipped states are grayed out in Figure 3.1. Three paths are explored as a result. We consider the range $[\tau, \tau')$ as a half-open interval where the start is considered part of the range but not the end. Thus we produce two test cases as a result.

Figure 3.1 Symbolic execution between paths ρ and ρ'.



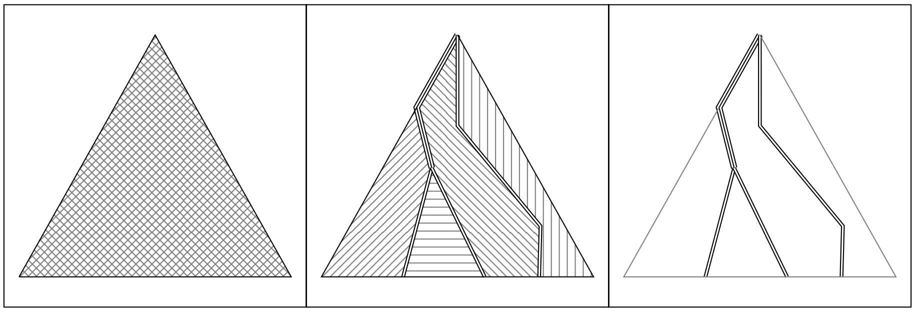
Once we have the basic mechanism for ranged symbolic execution, we use it in three novel ways:

- **Incremental execution:** Ranged symbolic execution enables symbolic execution to be paused at any stage and it can be restarted later with minimal overhead using the last input it generated as the start of new range.

- **Parallel execution:** Ranges of symbolic execution can be analyzed in parallel. For example, we can have three *non-overlapping* ranges for the above example `[null,`τ`)`, `[`τ`,`τ'`)`, and `[`τ'`,null)`. Executing these in parallel will completely analyze the above function without *any* communication between parallel nodes. Figure 3.2 shows a high level overview of dividing symbolic execution into non-overlapping ranges. Only the paths dividing the ranges are redundantly analyzed as path of both ranges. The initial set of dividing points can come from manual or random test cases or from symbolic execution of a previous version of code.

- **Parallel execution using work stealing:** We further provide an algorithm for parallel symbolic execution using work stealing when there is no initial set of inputs to form the ranges. For example, if a parallel node starts symbolic execution of the `mid` function and reaches the first branch `x<y`, it proceeds with the `true` branch while queueing the `false` branch in a list of work to be finished later. In the meanwhile, if another parallel node is free for work, it can steal work from the queue of this node and explore paths where `!(x<y)`.

Figure 3.2 High level overview of dividing symbolic execution into non-overlapping ranges for independent symbolic executions.



(a) Standard symbolic execution

(b) Ranged symbolic execution (4 ranges)

(c) Only boundaries redundantly analyzed

We next discuss using a single test input as analysis state and using two test inputs to define an analysis range (Section 3.1.2), performing symbolic execution within a range (Section 3.1.3), using ranged symbolic execution for parallel and incremental analysis (Section 3.1.4), and combining ranged symbolic execution with distributed work stealing for dynamic load balancing (Section 3.2.2.4).

3.1.2 Test input as analysis state

We introduce three concepts in this section: (1) describing analysis state with a single concrete test, (2) defining ordering of tests based on paths taken, and (3) using two concrete tests to define a range of analysis.

We introduce the concept of describing analysis state with a single concrete test.

Definition 1. *Given the ordering O of all paths ρ taken by a path-based analysis, any concrete test τ defines a state of analysis ς_τ where every path $\rho < \rho_\tau$ under O has been explored and none of the rest have been explored. ρ_τ is the path taken by test τ.*

Definition 1 assumes the existence of translation from concrete tests to paths and an algorithm to compare tests based an the ordering taken by the path-based analysis. The translation from concrete tests to paths can be done simply by executing the test and observing the path it takes. In practice, however, we will not need to find the corresponding path separately and it will be calculated along with other operations as discussed in the next section. Next, we discuss *test ordering*.

Definition 2. *Given two paths ρ and ρ', where $\langle \flat_0, \ldots, \flat_i \rangle$ is the set of basic blocks in ρ and $\langle \flat'_0, \ldots, \flat'_i \rangle$ is the set of basic blocks in ρ', we define that $\rho < \rho'$ if and only if there exists a k such that $\forall_{i=0}^{k} \flat_i = \flat'_i$ and the terminating instruction in $\flat_k$ is* *a conditional branch with $\flat_{k+1}$ as the "then" basic block and $\flat'_{k+1}$ as the "else" basic block.*

Definition 2 orders tests based on the paths they take. We find the first branch where the two paths differ. We consider the test taking the "true" side *smaller* than the test taking the "false" side. If two tests take the same path till the end, we consider them *equivalent*. Ordering more than two tests can be done by any sorting algorithm.

Definition 3. *Let τ and τ' be two tests with execution paths ρ and ρ' respectively, we define a range $[\tau, \tau')$ to be the set of all paths ρ_i such that $\rho \leq \rho_i < \rho'$.*

The benefit of defining a half-open range is that given three tests $\tau_a < \tau_b < \tau_c$, we have $[\tau_a, \tau_c) = [\tau_a, \tau_b) + [\tau_b, \tau_c)$.

We extend this concept to a set of n tests. We can find the paths taken by these tests and order them using the above algorithm. If the tests take p distinct paths ($p \leq n$), they define $p+1$ ranges of paths. Note that $p < n$ when multiple tests take the same path in code and are thus *equivalent*. The first and last range use special tests `begin` and `end`, where `begin` is the *smallest* path and `end` is one beyond the *biggest* path. The `end` is defined as one beyond the last path because we define ranges as half-open and we want all paths explored.

Lemma 1. *Ranged analyses on a set of $n-1$ ranges $[\tau_1, \tau_2), ..., [\tau_{n-1}, \tau_n)$ explore the same set of paths as the ranged analysis on $[\tau_1, \tau_n)$.*

This dissertation presents algorithms to perform symbolic execution of a program using ranges defined by tests. It shows how to *efficiently* perform symbolic execution of only the paths within a range. The next subsection presents these algorithms.

3.1.3 Ranged symbolic execution

In this section, we apply the technique of defining ranges of path-based analysis to symbolic execution. We call this *ranged symbolic execution*.

Definition 4. *Let* τ *and* τ' *be two tests that execute paths* ρ *and* ρ' *where* $\rho < \rho'$. *Define ranged symbolic execution for* $[\tau, \tau')$ *as symbolic execution of all paths* ρ_i *such that* $\rho \leq \rho_i < \rho'$.

Performing ranged symbolic execution has two parts: (1) defining the ranges given a set of tests and (2) executing each range of tests symbolically.

To define ranges given a set of tests, we can use any sorting algorithm given a comparator to compares two tests. Two tests can be compared either by running them independently and analyzing the branches they take or we can analyze two paths simultaneously until we find the first difference. The benefit of the second technique is that we only need to execute the common part of two paths and not explore two complete paths.

Algorithm 3.1 gives the algorithm for analyzing the common part of paths taken by two tests. The algorithm depends on a predicate function that checks if a given test satisfies a given condition. For that, we symbolically evaluate the path condition for the values in the given test. Note that checking if a path condition is satisfied by a given input is a very efficient operation. In contrast, we need much more time to solve a path condition to generate concrete test inputs using a SAT solver.

To run symbolic execution *between* two tests, we have to (1) convert them into paths, (2) find all paths between them, and (3) execute those paths symbolically. We do all three tasks simultaneously and thus we have no intermediate storage requirements.

Algorithm 3.1: Algorithm to compare two tests. This can be used with any sorting algorithm to order any number of tests.

```
input  : test $\tau$, test $\tau'$
output: BIGGER, SMALLER, or EQUIVALENT
define path-cond $\rho$, address-space AS, address-space AS';
$i$ = first instruction in func;
repeat
    if $i$ is-a conditional branch then
        cond $\leftarrow$ condition of $i$;
        if PathTakenByTest($\tau$, $\rho \wedge$cond, AS) then
            if NOT PathTakenByTest($\tau'$, $\rho \wedge$cond, AS') then
                return BIGGER;
            end
            $\rho \leftarrow \rho \wedge$ cond;
            $i \leftarrow$ first instruction in then basic block;
        else
            if PathTakenByTest($\tau'$, $\rho \wedge$cond, AS') then
                return SMALLER;
            end
            $\rho \leftarrow \rho \wedge$ NOT(cond);
            $i \leftarrow$ first instruction in else basic block;
        end
    else
        update AS for $i$ using $\tau$;
        update AS' for $i$ using $\tau'$;
        $i \leftarrow$ successor of $i$;
    end
until $i$ is the last instruction;
return EQUIVALENT;
```

Symbolic execution state for a particular path contains the set of path constraints and address space. At branches, the state is split into two states. States to be visited in the future are added to a queue of states. The order of choosing states from the queue determines the search strategy used. We use depth first search in this work.

For restricting symbolic execution to a range, we introduce new variables `startTest` and `endTest` in the symbolic execution state. The initial state gets the `startTest` and `endTest` parameters from program input. If one of the parameters is the special *begin* or *end* symbol, we just use `null` in its place. We perform symbolic execution normally while using the special HANDLEBRANCH function in Algorithm 3.2 for conditional branches.

Algorithm 3.2 works by checking if the current state has a `startTest` assigned *and* the `startTest` does not satisfy the branch condition. Since we defined test ordering with *true* branches preceding *false* branches, we need to eliminate the *true* branch from the search. Similarly if we have an `endTest` which *does* satisfy the branch condition, we eliminate the *false* branch from being explored.

3.1.4 Parallel and incremental analysis

Ranged symbolic execution enables parallel and incremental analysis. For parallel analysis, we take a set of tests and use them to divide the symbolic analysis into a number of ranges. These ranges are then evaluated in parallel. We can use more ranges then available workers so that workers that finish quickly can pick another range from the work queue. The initial set of tests can come from manual tests, a symbolic execution run on a previous version of code, or even from a shallow symbolic execution run on the same code. In our evaluation, we pick random collection of tests from a sequential run and use it to define ranges for the parallel run. In the next section, we introduce another way to parallelize that requires no initial set of tests. It uses work stealing to get to-be-explored states from a busy worker to a free worker, and in doing so, dynamically redefining the ranges for both workers.

Algorithm 3.2: Algorithm for handling a branch for ranged symbolic execution. Each state works within a range defined by a start test τ_{start} and an end test τ_{end}. A new state is created using a basic block to start execution from, and a pair of tests to define the range.

```
input  : state, branch, test $\tau_{start}$, test $\tau_{end}$
output: set of states to be explored

cond ← branch condition of branch;
$BB_{then}$ ← then basic block of branch;
$BB_{else}$ ← else basic block of branch;
if $\tau \wedge \neg(\tau_{start} \Rightarrow cond)$ then
    return {new state($BB_{else}$, $\tau_{start}$, $\tau_{end}$)};
end
if $\tau' \wedge \tau' \Rightarrow cond$ then
    return {new state($BB_{then}$, $\tau_{start}$, $\tau_{end}$)};
end
if cond is unsatisfiable then
    return {new state($BB_{else}$, $\tau_{start}$, $\tau_{end}$)};
else if ¬cond is unsatisfiable then
    return {new state($BB_{then}$, $\tau_{start}$, $\tau_{end}$)};
else if both are unsatisfiable then
    // triggers for unreachable code;
    return $\emptyset$;
else
    return {new state($BB_{then}$, $\tau_{start}$, null), new state($BB_{else}$, null,
    $\tau_{end}$) };
end
```

Ranged symbolic execution also enables *resumable* execution, where we can stop symbolic execution at any point and resume it by giving it the last generated test as the starting point. To use it in combination with parallel analysis, we would also need the original ending point of the paused range. In the evaluation, we show a scheme, where pre-defined ranges are analyzed in increments resulting in negligible overhead and greater flexibility.

Ranged symbolic execution can potentially be extended to support symbolic execution on incremental changes in code. This would require finding the *smallest* and *largest* paths leading to changed code and defining a range using them. We plan to present this technique in future work.

3.1.5 Dynamic range refinement

Dynamic range refinement enables dynamic load balancing for ranged symbolic execution using work with work stealing. It starts with a single worker node responsible for the complete range $[a,c)$. Whenever this node hits branches it explores the *true* side and puts the *false* side on a queue to be considered later. As other workers come, they can steal work from this queue. The state on the queue is persisted as a test case b and the range is redefined to $[a,b)$. The stolen range $[b,c)$ is taken up by another worker.

Our implementation of distributed symbolic execution using work stealing utilizes a master coordinator node and uses MPI for communication. Algorithm 3.3 gives the algorithm for work stealing coordinator. It maintains lists for waiting workers and busy workers. Whenever a node needs work it tries to find a busy worker and tries to steal work. If a previously started stolen work request completes, it passes the work to a waiting worker. Sometimes, a stolen work request fails because the node is already finished or there is no work in the queue at that time. In such a case, it tries to steal work from another worker node.

Algorithm 3.4 is the algorithm for a worker node. When it receives a range from the coordinator, it performs ranged symbolic execution on it. If it receives a request to steal work, it checks if there is any state in the work queue. If so, it converts it to a concrete test to easily pass to the coordinator, and redefines the current symbolic execution range to end at that test. If there is no state in the work queue, it informs the coordinator of a failure. The worker repeats getting work and stealing ranges until the coordinator tells it to shut down.

Using intermediate states in this manner is different from using concrete tests that represent complete paths in code (like Section 3.1.4). Intermediate states, on the other hand, represent partial paths. Partial paths can result in overlapping ranges and more work that absolutely necessary. We circumvent this by choosing zero values for any fields not accessed by the concrete test. This extends the partial path to make a complete path that satisfies a zero value assignment for remaining fields. It is possible that such a path ends up being infeasible, but it is a complete path and sufficient to define non-overlapping symbolic execution ranges.

3.1.6 Evaluation

To evaluate ranged symbolic execution, we pose the following research questions:

- If we divide a symbolic execution problem into ranges and execute them sequentially, how does the performance compare with respect to running symbolic execution without ranges?

- If we divide a symbolic execution problem into ranges using some set of manual tests or tests from some prior iteration, and test those ranges in parallel on

Algorithm 3.3: Algorithm for work stealing coordinator.

```
define lists of waiting workers and busy workers;
count of workers with no theft started = 0;
give the whole task to the first worker;
while true do
    receive message $m$ from worker $w$;
    if $m$=need work then
        find a worker $w_2$ where no theft has been initiated; if no such process then
            increment count of workers with no theft started;
            if this count = total number of workers then
                terminate, we are done;
            end
        else
            ask $w_2$ to give stolen work;
        end
        add $w$ to list of waiting workers;
    else if $m$=stolen work then
        give stolen work to a waiting worker $w_2$;
        remove $w_2$ from list of waiting workers;
        if count of workers with no theft started $> 0$ then
            ask $w_2$ to give stolen work (again);
            decrement count of workers with no theft started;
        end
    else if $m$=cant steal then
        choose another busy worker $w_2$;
        ask $w_2$ to give some stolen work;
    end
end
```

Algorithm 3.4: Algorithm for work stealing worker node performing ranged symbolic execution.

```
while true do
    receive message m from coordinator;
    if m=exit then
        terminate;
    end
    else if m=new work then
        start ranged symbolic execution of new work ;
    else if m=steal work then
        if stealable states exist in symbolic execution state then
            remove state and convert it to a concrete test;
            send the concrete test to coordinator;
            update the end of current symbolic execution range;
        else
            inform coordinator that stealing failed
        end
    end
end
```

different machines, do we get a reasonable speedup?

- If we let a set of workers solve a symbolic execution problem without prior division and let them divide the problem dynamically using work stealing, how does the speedup compare to static ranges that require no communication?

In the following subsections, we describe (1) the set of test programs we use, (2) our methodology, (3) the experimental results, and (4) the threats to validity.

3.1.6.1 Subjects

To evaluate ranged symbolic execution, we use GNU core utilities or simply Coreutils[1] — the basic file, shell, and text manipulation core utilities for the GNU operating system.

Coreutils are medium sized programs with two to six thousand lines of code. Some of these programs do a particular task with a lot of error checks and thus form a deep search tree while others perform multiple functions and form a broad search tree. Deep trees are bad for any kind of parallel execution while broad trees provide opportunity for efficient parallel analysis. Considering all these utilities provides a good mix and representation of programs where parallelism in symbolic execution can and cannot help.

Coreutils were also used in the evaluation of KLEE symbolic execution tool. As, we build ranged symbolic execution using KLEE, Coreutils provide a good benchmark. We ran each program in Coreutils for ten minutes and chose 71 utilities

[1] `http://www.gnu.org/s/coreutils`

that covered more than a hundred paths in this time. Given more time, ranged symbolic execution can be evaluated for the other utilities as well.

3.1.6.2 Methodology

In this section, we discuss our test environment, how we ensure that all schemes cover the same paths for a quantitative comparison, how we define static ranges, and our experiments for work stealing.

We performed the experiments on the Lonestar Linux cluster at the Texas Advanced Computing Center (TACC). TACC enable reliable experiments as processors are completely allocated to one job at a time.

To compare standard symbolic execution with ranged symbolic execution and multiple parallel runs using these ranges, we want *every* technique to cover the same paths. To ensure this, we store the last path completely covered by standard symbolic execution and use this path as the bound for ranged executions. The time of standard symbolic execution shown in the tables is calculated from the start of execution to when this last completed path was covered. This adjustment provides a fair comparison between our standard and ranged symbolic execution.

We store generated tests from standard symbolic execution and choose nine random tests to define ten ranges for ranged symbolic execution. The end of the last range is fixed to the last test generated by standard symbolic execution (as discussed above). As the performance of ranged symbolic execution depends on the random tests chosen, we repeat the random selection and ranged symbolic execution five times and find the minimum, maximum, and average. We also find the minimum, maximum, and average times for the range taking the longest time for each set. This gives us the time for parallel execution using static ranges with 10 workers.

We then use 10 workers and 1 coordinator processor to symbolically execute the same problem with no a priori division using load balancing with dynamic work stealing. This experiment is not repeated multiple times as there is no non-deterministic choice of ranges to be made. All ranges are dynamically formed.

Lastly, we choose 5 programs that gave the worst speedup with parallel symbolic execution using work stealing, 5 programs that gave median speedup, and 5 programs that gave the best speedup. We use these 15 programs and run parallel symbolic execution using 5 and 20 workers also to see how speedup changes with number of available workers.

3.1.6.3 Experimental results

Table 3.1 shows the results for all 71 programs we tested. The second column has time for sequential symbolic execution using KLEE. The third column gives the minimum, maximum, and average times for covering the same paths divided into 10 random ranges. The fourth column has the minimum, maximum, and average time for the range taking the most time using the same ranges. Note that while the total time is pretty close for different random ranges, the time for the range taking the most time varies a lot. Thus the benefit of running in parallel depends on how good a static range is. This restriction applies to other parallel schemes as well that use static partitioning, e.g. [103]. The next column shows the calculated range of speedup achieved. The last two columns have the time and speedup for 11 processors (10 workers and 1 coordinator) when performing parallel symbolic execution using work stealing. We chose 10 workers so that the times can be directly compared to the times for10 parallel workers using random static ranges (column 4).

Table 3.1. Ranged symbolic execution for incremental and parallel checking for 71 program from GNU Coreutils suite of Unix utilities. At times the speedup is greater than 10X because of optimal use of caches in KLEE. KLEE is more efficient at solving multiple smaller problems than a single large problem.

	Standard	Incremental sym-	Parallel symbolic execution using 10 workers			
	symbolic	bolic execution	using static random ranges		using work	
Program	execution	time (s)	time (s)		stealing	
Name	time (s)	min / avg / max	min / avg / max	speedup	time (s)	speedup
base64	600	365 / 377 / 388	68 / 100 / 119	5.0 - 8.8X	83	7.2X
basename	156	110 / 115 / 126	18 / 32 / 63	2.5 - 8.6X	47	3.3X
cat	600	465 / 497 / 518	114 / 175 / 247	2.4 - 5.3X	90	6.6X
chcon	596	401 / 438 / 479	233 / 251 / 283	2.1 - 2.6X	193	3.1X
chgrp	569	283 / 301 / 325	68 / 138 / 175	3.3 - 8.4X	41	13.8X
chmod	550	243 / 256 / 267	73 / 78 / 88	6.2 - 7.6X	46	12.0X
chown	598	263 / 283 / 300	64 / 87 / 120	5.0 - 9.4X	41	14.4X
chroot	599	358 / 393 / 414	102 / 151 / 238	2.5 - 5.8X	330	1.8X
comm	607	730 / 929 / 1125	338 / 472 / 599	1.0 - 1.8X	630	1.0X
cp	600	231 / 264 / 290	58 / 120 / 175	3.4 - 10.3X	56	10.8X
csplit	601	349 / 366 / 387	105 / 162 / 196	3.1 - 5.7X	57	10.5X
cut	600	427 / 442 / 465	144 / 171 / 221	2.7 - 4.2X	105	5.7X
date	278	252 / 260 / 275	83 / 113 / 130	2.1 - 3.3X	84	3.3X
dd	601	353 / 379 / 402	121 / 162 / 195	3.1 - 5.0X	278	2.2X
df	341	151 / 153 / 154	38 / 59 / 69	5.0 - 8.9X	49	7.0X
dircolors	600	460 / 468 / 485	101 / 147 / 198	3.0 - 5.9X	113	5.3X
dirname	618	628 / 701 / 758	377 / 534 / 616	1.0 - 1.6X	574	1.1X
du	601	482 / 540 / 578	134 / 180 / 232	2.6 - 4.5X	115	5.2X
echo	600	400 / 419 / 441	112 / 156 / 203	3.0 - 5.3X	101	6.0X

Name	time (s)	min / avg / max	min / avg / max	speedup	time (s)	speedup
env	600	492 / 503 / 512	114 / 171 / 236	2.5 - 5.3X	116	5.2X
expand	600	334 / 352 / 367	60 / 110 / 169	3.6 - 10.1X	59	10.2X
factor	609	609 / 622 / 640	93 / 156 / 185	3.3 - 6.5X	540	1.1X
fmt	601	743 / 781 / 826	142 / 176 / 215	2.8 - 4.2X	255	2.4X
fold	600	216 / 227 / 246	62 / 73 / 83	7.3 - 9.7X	45	13.3X
ginstall	596	429 / 451 / 500	105 / 163 / 232	2.6 - 5.7X	281	2.1X
groups	588	658 / 667 / 686	130 / 169 / 214	2.7 - 4.5X	350	1.7X
head	600	229 / 282 / 380	42 / 111 / 246	2.4 - 14.3X	85	7.1X
id	600	257 / 270 / 293	104 / 125 / 140	4.3 - 5.7X	49	12.3X
join	594	499 / 530 / 582	108 / 131 / 162	3.7 - 5.5X	192	3.1X
kill	600	207 / 214 / 223	43 / 65 / 107	5.6 - 13.9X	76	7.9X
ln	600	179 / 213 / 255	38 / 99 / 166	3.6 - 16.0X	53	11.4X
mkdir	596	605 / 735 / 847	259 / 313 / 400	1.5 - 2.3X	474	1.3X
mknod	609	549 / 790 / 1134	485 / 555 / 662	0.9 - 1.3X	572	1.1X
mktemp	600	352 / 375 / 402	197 / 212 / 256	2.3 - 3.1X	240	2.5X
mv	598	438 / 482 / 601	257 / 305 / 335	1.8 - 2.3X	353	1.7X
nice	600	254 / 299 / 368	80 / 153 / 255	2.3 - 7.5X	64	9.4X
nl	600	253 / 285 / 330	72 / 141 / 210	2.9 - 8.3X	53	11.3X
nohup	601	323 / 365 / 422	107 / 185 / 276	2.2 - 5.6X	290	2.1X
od	601	609 / 637 / 654	120 / 209 / 264	2.3 - 5.0X	122	4.9X
paste	600	380 / 397 / 433	85 / 130 / 206	2.9 - 7.1X	83	7.3X
pathchk	599	313 / 364 / 442	100 / 169 / 208	2.9 - 6.0X	178	3.4X
pinky	600	173 / 198 / 227	47 / 67 / 81	7.4 - 12.7X	46	13.1X
pr	601	538 / 580 / 606	93 / 169 / 237	2.5 - 6.4X	108	5.6X
printenv	588	337 / 549 / 749	96 / 251 / 352	1.7 - 6.2X	46	12.8X
printf	598	188 / 219 / 273	53 / 81 / 121	4.9 - 11.3X	46	12.9X

Name	time (s)	min / avg / max	min / avg / max	speedup	time (s)	speedup
readlink	600	247 / 266 / 305	86 / 108 / 137	4.4 - 7.0X	41	14.7X
rm	603	344 / 375 / 392	109 / 148 / 194	3.1 - 5.6X	185	3.3X
runcon	598	227 / 252 / 280	54 / 86 / 141	4.2 - 11.2X	55	10.8X
seq	600	287 / 312 / 333	90 / 110 / 133	4.5 - 6.6X	105	5.7X
setuidgid	600	507 / 552 / 623	95 / 156 / 206	2.9 - 6.3X	253	2.4X
sha1sum	600	312 / 324 / 332	72 / 111 / 144	4.2 - 8.4X	70	8.6X
shred	600	334 / 397 / 452	96 / 154 / 203	2.9 - 6.3X	95	6.3X
shuf	600	338 / 358 / 380	82 / 114 / 142	4.2 - 7.3X	74	8.1X
split	600	496 / 513 / 524	134 / 206 / 254	2.4 - 4.5X	123	4.9X
stat	599	246 / 268 / 290	73 / 88 / 104	5.8 - 8.2X	79	7.6X
stty	601	154 / 170 / 183	37 / 49 / 74	8.2 - 16.5X	63	9.6X
su	418	331 / 340 / 348	115 / 134 / 143	2.9 - 3.6X	300	1.4X
sum	600	240 / 282 / 340	86 / 136 / 204	2.9 - 7.0X	52	11.5X
tac	602	381 / 480 / 579	210 / 313 / 406	1.5 - 2.9X	160	3.8X
tail	600	349 / 369 / 397	102 / 152 / 204	2.9 - 5.9X	81	7.4X
tee	600	280 / 306 / 336	84 / 128 / 207	2.9 - 7.1X	50	12.0X
touch	561	312 / 333 / 371	81 / 115 / 157	3.6 - 7.0X	282	2.0X
tr	597	497 / 638 / 730	395 / 459 / 583	1.0 - 1.5X	569	1.0X
tsort	600	541 / 545 / 551	113 / 153 / 189	3.2 - 5.3X	121	5.0X
tty	588	517 / 530 / 556	174 / 222 / 308	1.9 - 3.4X	294	2.0X
uname	599	156 / 194 / 230	31 / 71 / 109	5.5 - 19.3X	34	17.7X
unexpand	600	508 / 528 / 541	102 / 148 / 196	3.1 - 5.9X	121	5.0X
uniq	600	370 / 391 / 430	119 / 150 / 175	3.4 - 5.0X	58	10.3X
vdir	596	377 / 440 / 553	162 / 263 / 425	1.4 - 3.7X	125	4.8X
wc	600	555 / 570 / 591	109 / 136 / 187	3.2 - 5.5X	125	4.8X
who	600	304 / 332 / 377	69 / 123 / 225	2.7 - 8.8X	70	8.6X

Name	time (s)	min / avg / max	min / avg / max	speedup	time (s)	speedup
Average	581	371 / 409 / 454	117 / 166 / 220	3.3 - 6.8X	160	6.6X

Our speedup for parallel symbolic execution using work stealing ranges from 1.0X (no speedup) to 17.7X. As 17.7 is more than the number of workers, we investigated and found that KLEE uses a lot of internal caches which perform much better when they are of a reasonable size. Thus KLEE is more efficient at solving smaller problems that one bigger problem. This is intuitive as symbolic execution maintains a lot of internal state and memory maps and search operations are common. These search operations become much more efficient for smaller problems (with or without cache). Thus ranged symbolic execution often makes KLEE faster even when all ranges are executed sequentially.

Figure 3.3 contains a plot of the speedup of all 71 utilities ordered by the speedup achieved using work stealing. The line graph shows the speedup for parallel symbolic execution using work stealing, while the vertical lines show the range of speedup for parallel symbolic execution using static random ranges. The dot on the vertical line shows the average speedup for static ranges. Note that for one third of the subject programs, work stealing gives a speedup similar to the minimum speedup achieved using static ranges while for the other two third subject programs, its about the maximum speedup achieved using fixed ranges or even more. We believe that the first set of programs have narrow and deep trees while the second set of programs have broad trees that expose parallelism. This parallelism is extracted best using work stealing.

Figure 3.3 Speedup with 10 worker nodes using ranged symbolic execution for 71 program from GNU Coreutils suite of Unix utilities. Vertical bars show the range of speedup achieved using different random static ranges with the average pointed out. The line shows the speedup achieved using dynamic load balancing using work stealing.

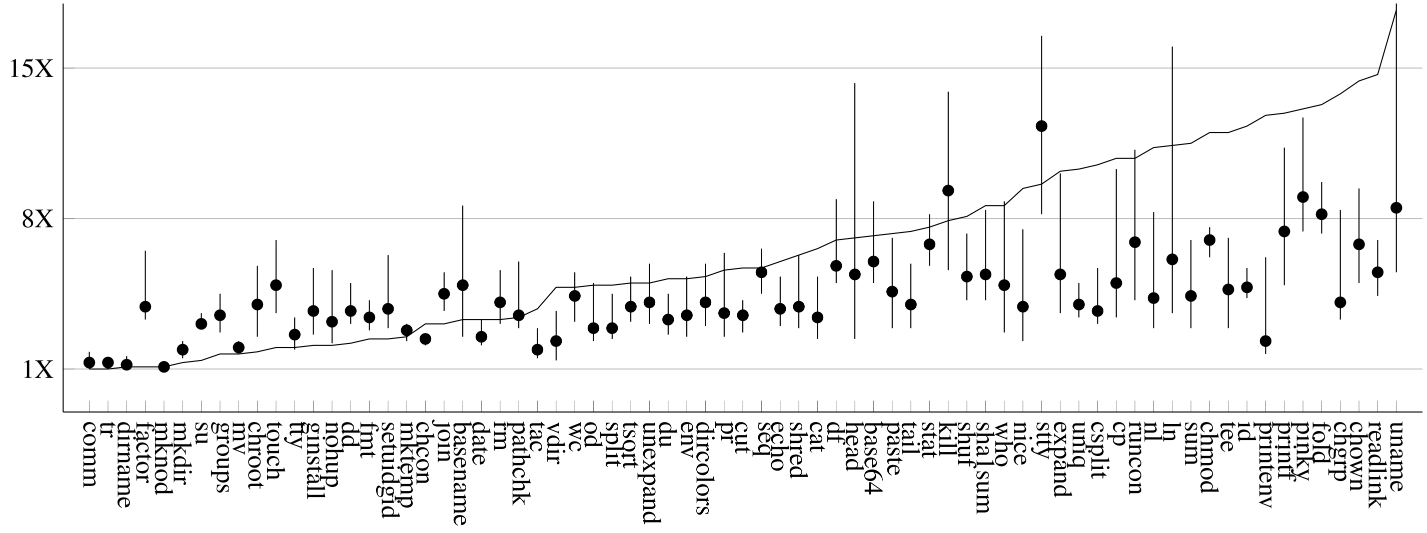
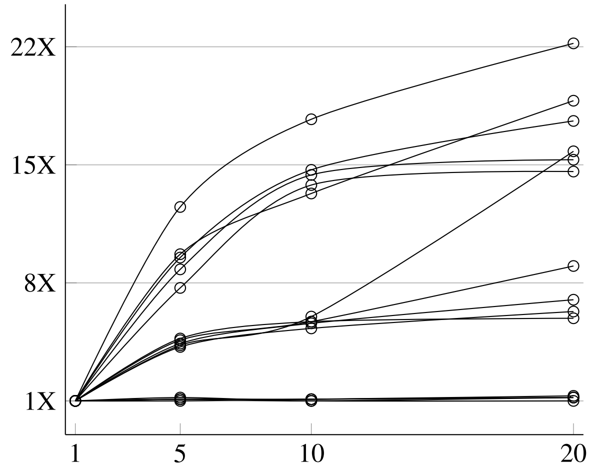


Table 3.2 Ranged symbolic execution with work stealing for 15 programs from GNU Coreutils on different number of workers. The +1 designates a separate coordinator node. These are the worst 5, median 5, and best 5 utilities from Figure 3.3 based on performance on 10 workers.

Program	Serial	5+1p		10+1p		20+1p	
Name	time(s)	time(s)	speedup	time(s)	speedup	time(s)	speedup
comm	607	573	1.1X	630	1.0X	514	1.2X
tr	597	509	1.2X	569	1.0X	595	1.0X
dirname	618	567	1.1X	574	1.1X	526	1.2X
factor	609	557	1.1X	540	1.1X	482	1.3X
mknod	609	593	1.0X	572	1.1X	505	1.2X
dircolors	600	142	4.2X	113	5.3X	95	6.3X
pr	601	138	4.4X	108	5.6X	86	7.0X
cut	600	130	4.6X	105	5.7X	67	9.0X
seq	600	129	4.7X	105	5.7X	102	5.9X
echo	600	139	4.3X	101	6.0X	38	15.8X
fold	600	62	9.7X	45	13.3X	32	18.8X
chgrp	569	74	7.7X	41	13.8X	39	14.6X
chown	598	68	8.8X	41	14.4X	39	15.3X
readlink	600	63	9.5X	41	14.7X	34	17.6X
uname	599	48	12.5X	34	17.7X	27	22.2X

Table 3.2 shows the results of running work stealing based ranged symbolic execution on a smaller set of 15 programs using 5, 10, and 20 workers with 1 coordinator processor and compares it to the performance of analyzing sequentially. Data for 1 and 10 processors is taken from Table 3.1. This data is plotted in Figure 3.4. These are the 5 worst, 5 median, and 5 best performing programs in the first experiment as discussed in Section 3.1.6.2. The 5 programs that performed worst in the first experiment do not gain anything from more processors and hardly give any further speedup. Most of the other 10 programs, however, gained more speedup. How much parallelism can be extracted from the symbolic execution of a program eventually depends on the program itself. If a program has a deep and narrow tree (e.g., one main path and only branching for error checks), than one or a few paths take nearly as much time as the time for complete analysis. Any scheme that completely checks one path on one processor cannot improve performance of such programs.

Figure 3.4 Time taken by 15 programs from GNU Coreutils on different number of workers for ranged symbolic execution with work stealing. These are the worst 5, median 5, and best 5 utilities from Figure 3.3 based on performance on 10 workers. The worst 5 overlap at or near 1.0X and are hard to distinguish.



3.1.6.4 Threats to validity

We tested our technique on one set of programs. It is possible that other programs exhibit different behavior. We mitigate this threat by choosing a suite of medium sized programs and then considering all of them. This is apparent in the results where we achieve a speedup of 1X (no speedup) to over 17X.

We selected random paths as range boundaries. We expect that in real scenarios, it might be more meaningful to divide ranges using tests from some manual test suite. It is possible that such ranges from manual tests provide much worse or much better performance. We mitigate this threat by repeating the random selection multiple times and reporting the range of speedup in both Table 3.1 and Figure 3.3.

We compared performance by comparing time taken for a given number of paths. Another advantage of ranged symbolic execution is covering a larger number of paths in the same amount of time. Yet another advantage is covering a small number of paths starting at different starting points from random or manual tests. This can be used to symbolically execute some paths of very large programs and get high coverage. However, we do not experiment these scenarios. It is possible that ranged symbolic execution is not as useful in these scenarios as we think. We plan to investigate this in our future work.

3.2 Parallel symbolic execution using master/slave architecture

Our work on ranged symbolic execution uses the idea to represent the state of a symbolic execution run using a test input as the basis of a novel approach to parallel symbolic execution. We also developed an alternative technique, ParSym [94], that uses a master/slave architecture to develop a novel parallel algorithm for scaling symbolic execution using a parallel implementation. In every iteration ParSym explores multiple branches of a path condition in parallel by distributing them among available workers resulting in an efficient parallel version of symbolic execution. We compare techniques for keeping communication low by stopping distribution once small enough work items have been formed. Experimental results show that symbolic execution is highly scalable using parallel algorithms: using 512 processors, more than two orders of magnitude speedup are observed. While we use symbolic execution to demonstrate our technique, it directly applies to a variety of other dynamic analyses that utilize the program's control-flow.

ParSym supports the combined symbolic and concrete execution model [42, 91]. In this model, the given program is executed on an initial input (either random or some initial values like zeroes, null, empty string etc.). A path constraint of the execution is built (as for pure symbolic execution). Each constraint in the path condition represents a branch on a symbolic input. The last constraint is negated and the path condition is solved for new concrete inputs. This leads to the execution of the *not taken* side of the last branch statement. When no new branches are encountered and the last constraint has already been negated once, the exploration backtracks on the path constraint to explore previous branches.

Our insight into parallel symbolic execution is two-fold: One, symbolic execution requires the negation of several branch conditions, each of which must be explored; and two, the time to solve path conditions dominates the time to execute a path. Symbolic execution is, therefore, likely to benefit significantly from parallel algorithms, which can effectively distribute the workload among different workers.

To keep the work distribution from becoming a bottleneck, we compare techniques for stopping distribution and solving the remaining small problems locally at parallel workers. Because of the unpredictable nature of path exploration tree, we need to generate many more sub-problems than available workers to keep them busy most of the time.

3.2.1 Illustrative Example

We describe the combined symbolic and concrete execution technique [42, 91] and intuition for our parallelization technique in this section. We take as example a function that checks if a given sequence is *bitonic* or not. A bitonic sequence either increases then decreases, or decreases then increases. Monotonic sequences are bitonic as well. Thus $\langle 1,2,3,2,1 \rangle$ is a bitonic sequence as is $\langle 3,2,1,0,1 \rangle$. We use symbolic execution on the first implementation in Listing 3.1 and use the second implementation to check the results. When the results differ, we can manually inspect the input and determine which implementation is producing the wrong result.

We provide this function with an array of four symbolic integers. Symbolic Execution can exhaustively traverse this function (no depth, iteration, or time

Listing 3.1. Two implementations of a function to check if a given sequence is *bitonic*. We symbolically execute the first function and compare results with the second function.

```

int isBitonic( int a[], int size ) {
  int firstDec=size-1, firstInc=size-1;
  int lastDec=0, lastInc=0;

  for( int i=1; i<size; ++i )
    if( a[i-1] > a[i] ) lastDec=i;
    else if( a[i-1] < a[i] ) lastInc=i;

  for( int i=size-1; i>0; --i )
    if( a[i-1] > a[i] ) firstDec=i-1;
    else if( a[i-1] < a[i] ) firstInc=i-1;

  return lastInc <= firstDec || lastDec <= firstInc;
}

int isBitonic2( int a[], int size ) {
  int i=1;
  while( i<size && a[i-1] <= a[i] ) ++i;
  while( i<size && a[i-1] >= a[i] ) ++i;
  if( i==size ) return 1;

  i=1;
  while( i<size && a[i-1] >= a[i] ) ++i;
  while( i<size && a[i-1] <= a[i] ) ++i;
  return i==size;
}
```

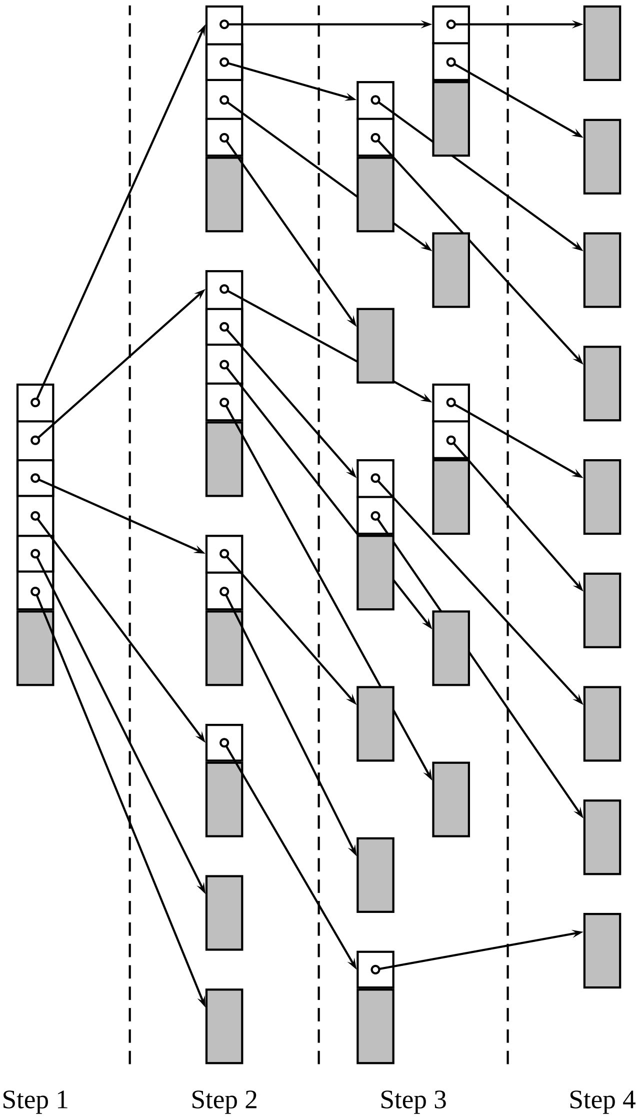
bound) in 27 iterations. In each run, the loops run 3 times each (one less than array size) and each loop has 3–6 conditions in it (based on whether the first *if* leads in to the *else* branch or not. Thus all path conditions for all paths consist of 6–12 constraints. Note that the comparisons in the *return* statement do not involve a symbolic variable or were assigned a symbolic variable and thus do not form a constraint on inputs.

The first iteration with all zeroes produces a path condition with 12 constraints. Symbolic execution takes the first constraint, negates it and solves it. It then reverts the first constraint and negates the second, solving the first two constraints. Then it negates the third (after reverting the second) and solves the first three constraints and so on. To solve a set of constraints, symbolic execution uses an external SAT solver. Six of the 12 set of constraints turn out to be unsolvable i.e. no solution. This can happen, for example, when two symbolic variables are required to be equal by one constraint and not equal by another. Solving the remaining six set of constraints produces six concrete outputs for future iterations.

The search space of symbolic execution is shown in Figure 3.5. Each vertical long box is a path condition resulting from one execution. It is divided into constraints. The arrow leading from a constraint connects it to the execution that resulted from negating this constraint and solving the set of constraints up to this constraint. Grey area represents one or more constraints whose negation produces an unsolvable set of constraints. Note that every path condition has the same initial set of constraints as its parent up to the constraint that was negated. These initial constraints are skipped over and not solved again. These are also not shown in the figure.

The most common search strategy of symbolic execution is depth first search. In this, only the first constraint is negated and solved. And then the concrete

Figure 3.5 Symbolic execution search tree for checking bitonic sequences of length 4. Each bar above represents a path constraint resulting from one execution of the program. Each white box is a constraint, negating which, produces a solvable formula and the arrow links it to the corresponding concrete execution. These 27 units of work can be done in parallel in only 4 steps.



inputs produces a new path constraint to be solved. Backtracking to a previous path condition only happens if new path conditions are fully explored (under given depth bound). Under any strategy a serial algorithm needs 27 steps to complete this work.

However we can see that the dependency chain is only 4 steps deep. These are separated by dashed lines in Figure 3.5. Thus, given enough processors (11 in this case), we can complete this task in only 4 steps instead of 27 steps. There is potential of 7x speedup[2] in this simple example. The parallelization potential for larger problems is even larger, and since there are numerous small tasks, workload can be neatly balanced between the processors.

3.2.2 Algorithm

Symbolic execution works with path conditions often stored as complex expression trees. The main problem in parallelizing symbolic execution is to avoid transport of these structures between parallel nodes. Concrete inputs, on the other hand, are usually small and can be further compressed by using input domains (predecided sets of values). We divide work such that each node gets a set of inputs, executes the program concretely while observing the symbolic path constraint built. It then negates each constraint turn by turn, solves the path condition for concrete inputs and sends the input indices to a parallel node. Thus we minimize communication overhead.

There are three parts of our parallel symbolic execution algorithm. The core symbolic execution engine that concretely runs programs, forms path conditions, and solves constraints, a central symbolic execution monitor that monitors the overall process, and a symbolic execution agent that helps the core engine communicate

[2] This number (7x) does not consider communication overhead, and that each task does not take equal time. It is given to show the potential of parallelization in the algorithm.

with the monitor. We describe these three components in the following subsections followed by work distribution optimization in Section 3.2.2.4 and a correctness argument in Section 3.2.2.5.

3.2.2.1 Symbolic execution engine

Our parallel symbolic execution algorithm is based on combined symbolic and concrete execution [42, 91]. We start by executing the program on an initial input (all zeroes, null pointers, empty strings etc.) and observe the path it traverses. We then negate the first constraint and solve it for concrete values. We then move to the next constraint, negate it, and solve all constraints up to the negated constraint in the path condition. We repeat until we get the specified number of solvable constraints (depth bound) or until all constraints have been used.

Baseline symbolic execution will re-run the program after solving the first constraint and then work on the new path condition formed (which should only differ in the part after the negated condition). It only backtracks to the original path condition when it has explored all newer path conditions exhaustively (under given depth bound).

In Algorithm 3.5, we show implementation of core engine in function PARSYM. It receives inputs and uses them for a concrete execution and observes the path condition like standard symbolic execution. It then checks the path condition from the `pos`th position and works until all constraints are traversed or enough new items are produced (according to depth bound). At each position the constraint is negated, if the resulting path condition (up to that constraint) is solvable, it produces a new work item that is at the given depth, whose constraints before `pos+1` have been negated once, and whose inputs are the result of solving.

We based our implementation an an open source symbolic execution imple-

Algorithm 3.5: Parallel Symbolic Execution.

```
input : workItem
(depth, pos, inputs) $\leftarrow$ workItem;
(constraints) $\leftarrow$ ExecAndObserve(testProgram);
while $depth>0 \land pos<Size(constraints)$ do
    Negate(constraints[pos]);
    (success, inputs) $\leftarrow$ Solve(constraints[0...pos]);
    if success then
        depth $\leftarrow$ depth-1;
        newWork $\leftarrow$ (depth, pos+1, inputs);
        NewWorkItem(newWork);
    end
    Negate(constraints[pos]);
    pos $\leftarrow$ pos+1;
end
```

mentation [12], the CIL instrumentation library [80], and the CVC3 SMT solver [6]. The Solve method in Algorithm 3.5 uses the CVC3 library to solve the constraint.

The key idea in this implementation is that the path constraints need not be transferred to other nodes. Only program inputs have to be transferred. Also the function proceeds from solving smaller (and therefore easier and quicker to solve) constraints to longer (and time taking) constraints. Thus the quicker a work can be dispatched to another node, the quicker it is done.

The initial work item contains the maximum depth specified by the user, index of first constraint in path condition to be negated (zero), nothing as input (zeroes and nulls are used as default by instrumentation). The function ExecAndObserve runs the instrumented program and returns the path condition. The path condition is used by rest of the algorithm.

3.2.2.2 Symbolic Execution Monitor

Symbolic Execution Monitor provides the central authority that carries out symbolic execution of a program. It starts by distributing the executable to be tested to all nodes running symbolic execution engines using agents running on each machine. After that it serves as master in a typical master slave configuration [68]. It contains a work queue of inputs to be used for symbolic execution. All other nodes have a symbolic execution engine that contacts the monitor for an input to process using the agent. Any new items generated by the engines are sent back to the monitor, and the monitor enqueues them.

The designation of one processor as monitor is helpful for a large number of processors. However for a small number of processors, it wastes a valuable resource. We anticipate that, if necessary, it can share a processor with an agent as a separate thread and thus avoid the cost for a small number of processors. However we do not implement this strategy.

The algorithm for monitor is given in Algorithm 3.6. The monitor maintains a list of agents that want to work (`agentQ`), agents that have no more work to do (`exitQ`), and a list of pending work items (`workQ`). This algorithm supports double buffering at the client (receiving the next work item while processing the previous one) and that's why two queues (`agentQ` and `exitQ`) are needed. When a agent has finished work and has not received new work in the background, it requests for addition to the `exitQ`.

Double buffering is important to minimize wasted time at the nodes performing actual symbolic execution. By the time they finish with one input, another is already ready for consumption. This eliminates the need to have larger work divisions.

Algorithm 3.6: Algorithm for symbolic execution monitor.

```
input  : agentCount, initialWorkItem
workQ $\leftarrow$ CREATEQUEUE( );
agentQ $\leftarrow$ CREATEQUEUE( );
exitQ $\leftarrow$ CREATEQUEUE( );
QUEUE(workQ, initialWorkItem);
while SIZE(agentQ) $\neq$agentCount $\wedge$ SIZE(exitQ) $\neq$ agentCount do
    (agent, cmd, workItem) $\leftarrow$ RECV(any);
    if cmd = QUEUE then
        if EMPTY(agentQ) then
            QUEUE(workQ, workItem);
        else
            agent$'$ $\leftarrow$ DEQUEUE(agentQ);
            SEND(agent$'$, WORK, workItem);
        end
    else if cmd = DEQUEUE then
        Remove(exitQ, agent);
        if EMPTY(workQ) then
            QUEUE(agentQ, agent);
        else
            workItem $\leftarrow$ DEQUEUE(workQ);
            Send(agent, WORK, workItem);
        end
    else if cmd = EXIT then
        QUEUE(exitQ, agent);
    end
end
forall the $s \leftarrow$ agentQ do
    SEND($s$, EXIT);
end
```

The initialWorkItem (underlined) for symbolic execution is an initial input that can be random or some predetermined initial values (like zeroes, nulls, empty strings etc.). It also contains depth of current symbolic execution iteration and position of the next constraint to be negated along with concrete inputs to the program. The initial input contains the depth bound and zeroth position to explore all parts of the constraint. The algorithm loops and serves messages from the agents.

There are three types of messages from agents to monitor:

- QUEUE: is used to ask the monitor to add new work items to the work queue.
- DEQUEUE: is used to ask the monitor for a new work item to process. If a work item is not readily available, the agent is remembered in agent queue. Whenever work items become available, they are sent to these free agents instead of being queued in work queue.
- EXIT: is used to tell the monitor that the agent has finished all work and is safe to exit. However this message does not mean that the agent *will* exit. In fact it may get more work from the monitor soon.

The monitor sends only two messages back to the agents:

- WORK: is used to give new work to the agent. It is sent in response to a DEQUEUE request so the agent should be ready and willing to receive it.
- EXIT: is used to ask the agent to exit. It is sent only when the agent has expressed will to exit by an EXIT message in the other direction. Therefore it should be able to safely exit immediately.

Safe termination of the algorithm assumes ordering of messages between monitor and a particular agent. Simultaneous occurrence of a agent in `exitQ` and agentQ requires that it send an EXIT message when it is in the agentQ (due to a previous DEQUEUE message). Any later DEQUEUE message would first remove it from exitQ. Also on a DEQUEUE message, the monitor immediately serves a agent if work is available. Thus simultaneous occurrence of *all* agents in both these queues means that no work is available in the workQ and all agents have finished their assigned work. At this time the monitor terminates its loop and sends an EXIT message to all agents.

3.2.2.3 Symbolic Execution Agent

Symbolic execution agents are responsible for providing work items to the core symbolic execution engine. Algorithm for agent processors is given in Algorithm 3.7. The two buffers workItem and workItemBack) allow double buffering. When one is being processed, data from the monitor is received in the other. This allows hiding communication overhead and latency and avoids any wasted time for the core engine.

At the start, every agent sends the monitor processor a DEQUEUE message followed by an EXIT message. If work is given to the agent, the EXIT message is benign (see Section 3.2.2.2). However if there are few work items, and the particular agent will never get any work to do, the pair of messages will allow the monitor to remember that this agent is free.

The agent loops until it receives WORK messages (an EXIT message will break the loop). To process a work item (inputs for symbolic execution), the function PARSYM in Algorithm 3.5 is used.This function traverses the search graph and finds new inputs for symbolic execution. Algorithm 3.7 also shows NEWWORKITEM, a wrapper for sending an item to the monitor.

Algorithm 3.7: Algorithm for agent processors for parallel dynamic analysis and helper functions for parallelizing specific dynamic analysis tools.

```
workItem $\leftarrow$ createBuffer( );
workItemBack $\leftarrow$ createBuffer( );

Send(monitor, DEQUEUE);
Send(monitor, EXIT);
(cmd, workItem) $\leftarrow$ Recv(monitor);
while cmd = WORK do
    Send(monitor, DEQUEUE) (cmd, workItemBack) $\leftarrow$
    RecvStart(monitor)

    ParSym(workItem);

    if not RecvFinish() then
        Send(monitor, EXIT);
        WaitForRecvFinish( );
    end
    workItem $\leftarrow$ workItemBack
end
input  : workItem
Send(monitor, QUEUE, workItem) (TODO);
```

3.2.2.4 Work Distribution Optimization

We compare three different techniques for reducing communication overhead while load balancing the work. Since it is not possible to predict the shape of state space, it is not possible to pre-divide the search space perfectly for independent parallel analysis. Thus, if work is divided such that every processor gets one task, load will be badly divided but there will be almost no communication. On the other hand, if each and every processing task is distributed centrally, load balancing will be perfect with high communication overhead. We evaluate the perfectly load balanced technique and compare it to two techniques that stop distributing work at a point and then leave the remaining problems to be solved locally. The three techniques are:

- **Load Balanced:** In this technique, every node explored in the search tree is sent to the central monitor which decides where to send it for processing.
- **Fixed Number of Sub-Problems per Agent:** In this technique, we keep dividing the work until a fixed number of problems (specified relative to the number of available agents) has been generated. For example, we can generate twice as many problems as agents. Then if some agent got a big problem, another with a small problem can solve three in that time.
- **Fixed Number of Work Queue items:** In this mode, we keep dividing work until the work queue has a given number of problems queued. If problems are being generated and other agents are taking them and solving them, the division continues. It only stops when work queue accumulates the given number of sub-problems.

3.2.2.5 Correctness

In this section we argue correctness of parallel symbolic execution. We define correctness as being able to generate the same concrete inputs given the same depth bound with no time or iteration bound. Given a time or iteration bound, parallel symbolic execution is not guaranteed to produce the same results even on repeated executions (with more than one agent). Also with respect to the usual depth first approach, parallel symbolic execution is considering inputs in a slightly different order (explained in Algorithm 3.5). Thus the only meaningful comparison is when there is no time or iteration bound, however a depth bound can be present.

Note first that parallel symbolic execution can never go deeper than standard symbolic execution. This is because when a work item is generated, the current depth is stored in it. When a agent processes this work item, it uses the depth stored in the item and ensures that the maximum depth condition is satisfied.

Now consider that the constraints negated and the set of constraints up to the negated constraint solved by the parallel version are the same as the serial version. The difference being that serial version solves one set of constraints, goes deep, then backtracks to solve the second set of constraints, and so on. Thus with no time or iteration bound, both versions will end up generating the executions.

3.2.3 Evaluation

We evaluated parallel symbolic execution on four problems. One is symbolic execution of GREP 2.2. The other is using symbolic execution to perform bounded exhaustive testing of binary trees. Third is our small example from Section 3.1.1 to illustrate path explosion problem and how parallelization mitigates it. And fourth is a function checking a condition on each element of a large linked list. We used Lonestar, a Linux cluster containing Xeon 2.66GHz processors with more than 5000 cores and an InfiniBand interconnect, graciously provided by Texas Advanced Computing Center (TACC). We tested our executions on up to 512 processors (maximum allowance). We also tested using 128, 32, and 8 processors. Lastly we tested on 2 processors to see the communication overhead (as one processor is doing useful work and one is acting as monitor). Serial time reported is measured by executing on a dedicated processor from the same cluster.

GREP 2.2 GREP 2.2 is a 15K lines of C code application and sufficiently complex due to processing of regular expressions and other search patterns on the input. We provided it with 10 symbolic characters to be searched within a string of 40 symbolic characters [12].

Bounded Exhaustive Testing Bounded exhaustive testing generates all valid test inputs within given bounds. This is not trivial for complex structures. Specialized solver Korat [10] provides means to perform bounded exhaustive testing for complex structures. CUTE [91] introduced support for pointers in symbolic execution and discussed the idea of performing bounded exhaustive testing for complex structures using CUTE. To perform this test, we added support for pointers to the open source CREST tool [12], which does not have built-in support for pointers, and generated all valid binary trees from size 5 to 10.

Parallel symbolic execution enables bounded exhaustive testing to process larger bounds and thus further increase the confidence on the program under test.

Bitonic Sequences We did exhaustive symbolic execution for this problem for arrays of varying number of symbolic integers. Due to the large number of possibilities of integer ordering, we have a great number of paths and face the path explosion problem. The usual solution is to depth bound it *and* possibly use some heuristics like no new coverage observed for some time in some direction. However in this experiment, we do want the number of paths to explode and observe how parallelization tames path explosion.

Secondly, we include this experiment because we discussed this basic example for motivation of parallelization. We showed the potential of parallelization in this simple example. Now we want to see if that potential can be realized by our implementation.

Linked List Lastly, we examined a function which checks some properties of each element of a large linked list. We consider this function as a degenerate example where parallelization does not help. This happens because of the shape of the search space for a linked list does not provide enough potential for parallelism.

Discussion Performance results for all our test subjects are given in Table 3.3. Due to the similar nature of performance results for our widely different test subjects, we discuss them together in this section. We make a few key observations:

- High speedups are achieved. We achieved up to 135x speedup for GREP 2.2 whereas an even higher 405x for bounded exhaustive testing. By looking at the last line in both tables, we can see that this would give an even higher speedup, but it cannot be measured due to TIMEOUT on serial execution. TIMEOUT is set to 5 hours. For generation of binary trees of 10 nodes, 512 processors finished the work in 90 seconds whereas serial processor timed out in 5 hours.

Table 3.3 Performance data for using parallel symbolic execution. For each parallel run, one processor acts as monitor and the rest as agents. TIMEOUT is set to 5 hours.

Iterations	Serial	Parallel Time (Speedup)				
	Time	2p	8p	32p	128p	512p
1,000	0:00:28	0:00:33	0:00:10 (3x)	0:00:08 (3x)	0:00:02 (13x)	0:00:04 (6x)
10,000	0:05:15	0:05:48	0:00:59 (5x)	0:00:18 (17x)	0:00:08 (41x)	0:00:04 (70x)
100,000	1:09:23	1:17:10	0:11:14 (6x)	0:02:37 (26x)	0:00:53 (78x)	0:00:31 (135x)
650,000	TIMEOUT	TIMEOUT	TIMEOUT	3:47:05	0:52:35	0:13:14

(a) Symbolic Execution of GREP 2.2

Iteration	Iterations done					
Limit	2p	4p	8p	32p	64p	128p
1p	2004	12491	49916	179963	256671	130090
2p	1862	18776	39380	109475	138130	257292
3p	1860	23901	41293	123728	251573	282853
4p	1858	18663	37749	179506	238349	329559

(b) GREP iterations in one minute when stopping work distribution after fixed number of sub-problems formed

Work Queue	Parallel Time (Speedup)					
Limit	2p	4p	8p	32p	64p	128p
10	1862	23855	37757	167782	231159	268284
20	1976	23592	32028	176958	234732	372953
30	1995	23870	35635	163517	265396	343421
40	2606	19499	33657	168369	316094	319200

(c) GREP iterations in one minute when stopping work distribution after work queue reaches given size

Size	Serial	Parallel Time (Speedup)				
	Time	2p	8p	32p	128p	512p
5	0:00:29	0:00:34	0:00:09 (4x)	0:00:08 (4x)	0:00:13 (3x)	0:00:10 (3x)
6	0:02:05	0:02:22	0:00:23 (5x)	0:00:07 (18x)	0:00:11 (11x)	0:00:12 (10x)
7	0:08:57	0:10:08	0:01:29 (6x)	0:00:26 (21x)	0:00:13 (42x)	0:00:19 (28x)
8	0:38:18	0:43:01	0:06:18 (6x)	0:01:24 (27x)	0:00:28 (83x)	0:00:23 (99x)
9	2:44:12	3:04:37	0:26:18 (6x)	0:06:04 (27x)	0:01:36 (102x)	0:00:24 (405x)
10	TIMEOUT	TIMEOUT	1:41:09	0:25:02	0:06:14	0:01:34

(d) Bounded Exhaustive Testing for Binary Trees

Size	Serial	Parallel Time (Speedup)				
	Time	2p	8p	32p	128p	512p
6	0:00:32	0:00:38	0:00:09 (3x)	0:00:06 (5x)	0:00:12 (3x)	0:00:09 (3x)
7	0:01:59	0:02:14	0:00:23 (5x)	0:00:10 (11x)	0:00:12 (10x)	0:00:11 (10x)
8	0:07:59	0:07:55	0:01:09 (7x)	0:00:20 (25x)	0:00:11 (43x)	0:00:09 (54x)
9	0:25:04	0:28:09	0:04:03 (6x)	0:01:01 (25x)	0:00:23 (66x)	0:00:11 (140x)
10	1:27:25	1:37:08	0:13:56 (6x)	0:03:14 (27x)	0:00:53 (99x)	0:00:17 (310x)
11	TIMEOUT	TIMEOUT	0:47:51	0:10:49	0:02:47	0:00:44

(e) Exhaustive Symbolic Testing for Bitonic Sequences

Size	Serial	Parallel Time (Speedup)		
	Time	2p	8p	32p
1000	0:00:10	0:00:12	0:00:15	0:00:13
3000	0:00:69	0:00:79	0:00:80	0:00:83
5000	0:03:15	0:03:36	0:03:41	0:03:42

(f) Function working on linked list elements

- Efficiency increases with problem size. For 8 processors, it ranged between 3–6x (7x maximum possible as there are 7 agents) and for 32 processors it ranged between 17–27x except for the smallest problems.

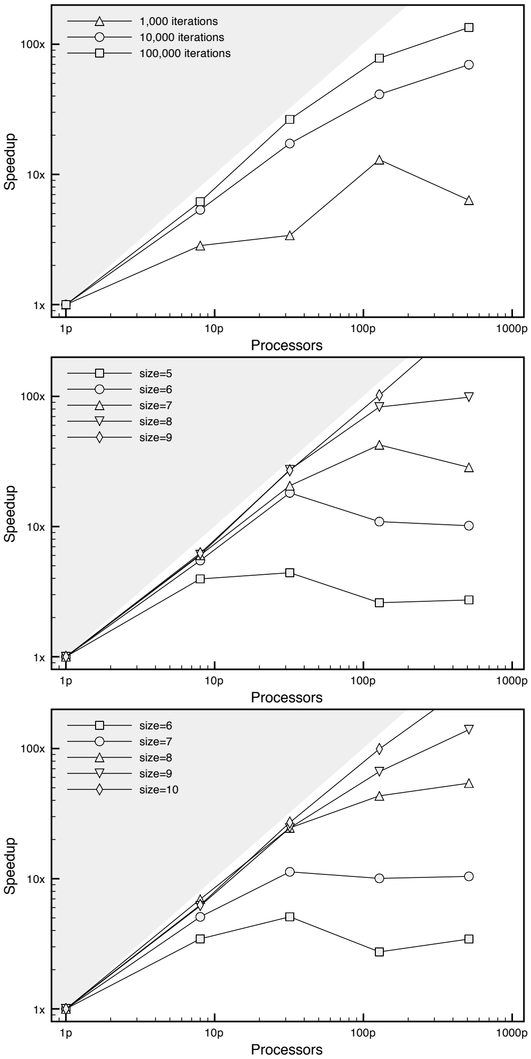
- There are problems where parallelism does not give enough benefit. We present the degenerate case of operations on linked lists where parallelism provides almost no benefit. However most programs have more parallelism than a degenerate linked list and can be exploited by our algorithm.

- Lastly, we observe that stopping distribution of work earlier has its benefits. However the exact parameters that work best are both problem development and on the number of agents. If the tree is much deeper, and the work division is stopped earlier, it can increase the overall time as some processors can be free.

We plot the number of processors versus speedup on a logarithmic scale. The plots for all three problems are shown in Figure 3.6. The grey area represents super-linear speedup. Since both scales are logarithmic, processors (1, 8, 32, 128, and 512) and their speedup are nicely spaced. The key information from this graph is that scalability improves with problem size. Smaller problem can deteriorate in performance when given too many processors. Also note that the best performance is close to linear. This means that given a big enough problem, we utilize the resources efficiently.

These results are highly encouraging and support our intuition in Section 3.1.1 of the potential of parallelism in symbolic execution. We believe that we were able to extract high degree of parallelism using our algorithm.

Figure 3.6 Plot of speedups versus number of processors used for all three test programs. Grey area is super-linear speedup. Both scales are logarithmic. (a) Symbolic Execution of GREP 2.2 (b) Bounded Exhaustive Testing for Binary Trees (c) Exhaustive Symbolic Testing for Bitonic Sequences.



Chapter 4

Constraint-driven analysis for black-box testing

This chapter describes the Pikse suite of techniques for improving constraint-driven analysis for black-box testing. We develop our techniques in the context of the Korat algorithm [10]. Specifically, we present multi-value comparisons (Section 4.1) and focused generation (Section 4.2), which allow more efficient and effective testing using Korat, as well as Parallel Korat (Section 4.3), which uses a master-slave architecture for parallel black-box test input generation. These techniques were initially presented at ICST 2012 [60], ASE 2009 [98], and ICST 2009 [95].

4.1 Multi-value comparisons

A key element of constraint-based testing is generation of test inputs from *input constraints*, i.e., properties of desired inputs, which is commonly performed by solving the constraints. We present a novel approach to optimize input generation from *imperative* constraints, i.e., constraints written as predicates in an imperative language. A well known technique for solving such constraints is execution-driven monitoring, where the given predicate is executed on candidate inputs to filter and prune invalid inputs, and generate valid ones. Our insight is that a lightweight static data-flow analysis of the given imperative constraint can enable more efficient solving. This dissertation describes an approach that embodies our insight and evaluates it using a suite of well-studied subject constraints. The experimental results show our approach provides substantial speedup over previous work.

This section focuses on constraints written as imperative predicates in C++, which due to its wide familiarity provides the basis of a framework that can be used by many developers.

Given the constraints, a key technical challenge in automating this methodology is efficient generation of *valid* inputs, which satisfy the given constraints, i.e., for imperative constraints, generating inputs for which the corresponding predicate returns true. For programs that operate on dynamically allocated data with complex structural properties, input generation can require costly exploration of very large input spaces to find valid inputs, e.g., searching the space of all strings (up to a certain length) to find strings that represent valid XML documents.

While Korat enables an efficient way to prune the input space, it still requires checking each candidate input that is not pruned using a complete execution of the given predicate. However, such executions can be wasteful, particularly on candidate inputs that are largely similar.

We provide a novel approach for more efficient solving of imperative predicates using a lightweight static data-flow analysis. Our insight is that repeated predicate executions can be optimized by performing certain comparison operations, which determine the predicate's output, against *sets* of candidate values for fields used in the comparisons, i.e., performing *multi-value* comparisons, rather than comparing individual values in turn as in standard execution. Thus, predicate executions on many candidate inputs that are similar are *forwarded* and the total execution cost reduced. Conceptually, our approach resembles stateful model checking where non-deterministic choice allows an expression to evaluate to different values, each of which is used in turn. However, a key difference is that we do not require storing and re-creating entire states. Moreover, our approach directly utilizes how field values determine the predicate's output in enumerating valid inputs as well as in pruning invalid ones.

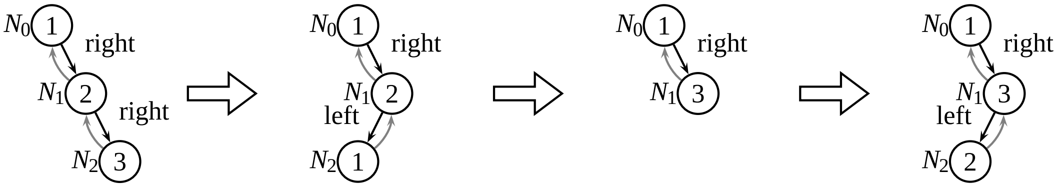
4.1.1 Illustrative example

Our approach, Korat$_{\text{multi}}$, uses multi-value comparisons based on lightweight data analysis. It is a technique for reducing the amount of `repOk` executions required to find all valid candidates by Korat. It is an automated technique that requires no modification to the `repOk` predicate.

Korat$_{\text{multi}}$ performs a static data-flow analysis on the LLVM bit-code of the program and instruments it. This analysis identifies any field accesses in `repOk` that are used in a comparison that *directly results* in the predicate failing or succeeding. Here, "directly result" means that the result of the comparison is either returned directly or it is used in a conditional branch that results in returning from the function on either the true side or the false side. Once such comparisons are identified, we instrument them with a special call with the field accessed, the comparison, and the other side of the comparison as arguments. This special call performs a multi-value comparison against all values in the field domain of this field. If, however, the field access is not used in such a comparison, we instrument it with a simple call to monitor field access (as done in standard Korat).

We explain the multi-value comparisons in Korat$_{\text{multi}}$ that are performed using this static data-flow analysis for the state of exploration shown in Figure 4.1(a). At this stage, the accessed fields are `(root,` N_0`.data,` N_0`.left,` N_0`.right,` N_1`.parent,` N_1`.data,` N_1`.left,` N_1`.right,` N_2`.parent,` N_2`.data,` N_2`.left,` N_2`.right, size)`. Like standard Korat, we backtrack over the first few fields until we reset N_1`.right` to `null` and set N_1`.left=`N_2. At

Figure 4.1 Four intermediate states when using Korat with multi-value comparisons for binary search trees of three nodes. All accessed fields are shown in "field+condition" column along with any condition deduced from static lightweight analysis. For each field, the values to be tried are shown in "values to try" column. Striked out values are "forwarded" based on the comparison. Current value is shown in square brackets.



field+condition	values to try	field+condition	values to try	field+condition	values to try	field+condition	values to try
root	0,[N_0]	root	0,[N_0]	root	0,[N_0]	root	0,[N_0]
N_0.data	[1],2,3	N_0.data	[1],2,3	N_0.data	[1],2,3	N_0.data	[1],2,3
N_0.left	[0],N_0,N_1	N_0.left	[0],N_0,N_1	N_0.left	[0],N_0,N_1	N_0.left	[0],N_0,N_1
N_0.right	0,N_0,[N_1]	N_0.right	0,N_0,[N_1]	N_0.right	0,N_0,[N_1]	N_0.right	0,N_0,[N_1]
N_1.parent=N_0	~~0~~,[N_0],~~N_1,N_2~~	N_1.parent=N_0	~~0~~,[N_0],~~N_1,N_2~~	N_1.parent=N_0	~~0~~,[N_0],~~N_1,N_2~~	N_1.parent=N_0	~~0~~,[N_0],~~N_1,N_2~~
$1<N_1$.data	~~1~~,[2],3	$1<N_1$.data	~~1~~,[2],3	$1<N_1$.data	~~1~~,2,[3]	$1<N_1$.data	~~1~~,2,[3]
N_1.left	[0],N_0,N_1,N_2	N_1.left	0,N_0,N_1,[N_2]	N_1.left	[0],N_0,N_1,N_2	N_1.left	0,N_0,N_1,[N_2]
N_1.right	0,N_0,N_1,[N_2]	N_1.right	[0],N_0,N_1,N_2	N_1.right	[0],N_0,N_1,N_2	N_1.right	[0],N_0,N_1,N_2
N_2.parent=N_1	~~0,N_0~~,[N_1],~~N_2~~	N_2.parent=N_1	~~0,N_0~~,[N_1],~~N_2~~	size=2	~~3~~	N_2.parent=N_1	[N_1]
$2<N_2$.data	~~1,2~~,[3]	$1<N_2$.data<2	~~1,2,3~~,			$1<N_2$.data<3	~~1~~,[2],~~3~~
N_2.left	[0],N_0,N_1,N_2					N_2.left	[0],N_0,N_1,N_2
N_2.right	[0],N_0,N_1,N_2					N_2.right	[0],N_0,N_1,N_2
size=3	[3]					size=3	[3]

this point, unlike standard Korat, the `repOk` execution in Korat$_{\text{multi}}$ will not fail because N_2`.parent=null` (its initial value). The access to N_2`.parent` would have been statically instrumented with a function call. This function call is invoked during `repOk` execution and that this is the first access to N_2`.parent`. Therefore, our optimization is applicable and we can do a multi-value comparison. We also get as arguments the comparison (equals) and the value to compare against (`null` in this case). We do a multi-value comparison of this value against all values in the field domain of N_2`.parent`. All invalid choices for N_2`.parent` (that would result in returning `null`) are forwarded over without backtracking and executing `repOk` on them again. The only valid choice is used for further execution and we reach Figure 4.1(b). This takes 11 `repOk` executions instead of 16 for standard Korat.

From there, we forward over N_2`.data` and N_2`.parent` fields to reach Figure 4.1(c). Lastly, we backtrack and try all fields while forwarding over `parent` and `data` fields to arrive at Figure 4.1(d).

This instrumentation based on static data-flow analysis and dynamically performing multi-value comparison in Korat$_{\text{multi}}$ using this information enables us to significantly reduce the number of `repOk` executions required while still producing all valid structures. For a binary search tree with three nodes, we find all five structures in Figure 2.1 on page 20 using 238 `repOk` executions for standard Korat. Korat$_{\text{multi}}$, on the other hand, requires only 173 `repOk` executions.

Next, we go over all the algorithms we employ to enable multi-value comparisons in Korat$_{\text{multi}}$. We describe (1) the high-level Korat$_{\text{multi}}$ algorithm and define *marked* and *unmarked* candidates, (2) the multi-value comparisons of a value against all values in a field domain that *mark* candidates, (3) the data-flow analysis that instruments the `repOk` predicate with calls to multi-value comparisons, and lastly (4) discussion of correctness with respect to the standard Korat algorithm.

4.1.2 The Korat$_{\text{multi}}$ Algorithm

The Korat$_{\text{multi}}$ algorithm builds upon the Korat algorithm by utilizing the information that some candidates are *marked*. A candidate is *marked* when the result of running `repOk` can be determined without running `repOk`. This determination comes using a combination of a multi-value comparison (Section 4.1.3) on the last accessed field and light-weight static data-flow analysis (Section 4.1.4). The data-flow analysis determines the correlation of the return value of the `repOk` predicate and the result of the multi-value comparison.

Standard Korat, after completing a `repOk` iteration and receiving field-access list, picks the last accessed field and chooses the next value from its field domain. On the other hand, Korat$_{\text{multi}}$ picks the next *unmarked* value from its domain. Marked values are either “known to succeed” meaning `repOk` *will* accept them or “known to fail” meaning `repOk` *will* reject them. Known to succeed values are included in successful candidates directly, while known to fail values are simply forwarded over. If there are no more unmarked values in the field domain, Korat$_{\text{multi}}$ backtracks to the field accessed before the last accessed field while clearing all markings on the last accessed field. This is done because the markings are only valid on one path.

Thus Korat$_{\text{multi}}$ divides candidates into marked and unmarked candidates. Unmarked candidates need a complete `repOk` execution, whereas marked candidates can be accept or forwarded-over without executing `repOk`. This forwarding results in much fewer `repOk` executions and a substantially lower execution time per candidate considered.

4.1.3 Multi-value comparisons

Korat$_{\text{multi}}$ depends on candidates being *marked* during the execution of `repOk` predicate. These markings are done by multi-value comparisons. A multi-value comparison compares a given value against all values in the field domain of a field not yet accessed in the `repOk` predicate. If a field has already been accessed, the given value can only be compared against its assigned value and no multi-value comparison can take place.

A multi-value comparison is *only* useful if at least one of the possible results (`true` an `false`) can determine what the `repOk` predicate will return. This information is gathered statically (Section 4.1.4).

The method that performs a multi-value comparison and determines if some candidate needs to be marked helps in forwarding candidates without running `repOk` and we thus call it `forwardFn` in this paper. The `forwardFn` method takes five arguments. Three of these arguments are for the multi-value comparison. They are (1) the given value to be compared (e.g. `2` in `if (size==2)`), (2) the comparison operator (==,<,> etc.), and (3) the field where values in its field domain are all compared against the given value. Two other boolean arguments are statically determined (Section 4.1.4) and inform if a `true` result of the comparison means that `repOk` will return `true` and if a `false` result of the comparison means that `repOk` will return `false`. For example, in "`if (size==2) return false;`", we *cannot* mark a candidate if the comparison results in a `true` value, but we *can* mark it to be forwarded if it results in a `false` value.

Algorithm 4.1 and Algorithm 4.2 gives `usefn` and `forwardFn` methods. The `useFn` method is invoked for every field access in standard Korat and it builds the field-access list. We also use it in Korat$_{\text{multi}}$ to instrument all accesses except those that classify as *forward-able* and are thus instrumented with `forwardFn`.

Algorithm 4.1: Algorithm for dynamic access monitoring (useFn)

```
input : Value v
if v is a controlled variable and not accessed before then
    add v to field-access list;
end
```

The `forwardFn` function checks if the given field has not been accessed before (like `useFn`). If not, it performs a multi-value comparison against all values in the field domain of this field. If the comparison is successful and `trueForward=true` (true side leads to returning constant `true`), it marks those values as accepted without even running `repOk` on them. Similarly, if the comparison is false and `falseForward=true` (false side leads to returning constant `false`), it marks them as skipped. If the initial value (as Korat initializes all fields to the first value in their field domains) of this field is marked as accepted or skipped during

Algorithm 4.2: Algorithm for dynamic access monitoring (forwardFn)

```
input  : Value v, Cond c, Value otherVal, trueForward, falseForward
if v is a controlled variable and not accessed before then
    forall the Values i in domain of v do
        if result of applying c on i and otherVal is true then
            if trueForward then
                mark i as accepted without running repOk;
            else if falseForward then
                mark i to be skipped (no need to run repOk);
            end
        end
        initialize v to first unmarked value;
    end
end
return result of applying c on v and otherVal
```

this process, it *forwards* to the first unmarked (not accepted, not skipped) value. If all values are marked, it chooses the last value and proceeds. These markings are used by the updated Korat algorithm to forward candidates without running `repOk`.

4.1.4 Data-flow analysis

Standard Korat monitors all field accesses made during `repOk` execution. Algorithm 4.3 shows monitoring and instrumentation of field accesses using LLVM [1]. The function `useFn` dynamically determines if this is the first access to this field; in which case it is added to field-access list [10].

Korat$_{\text{multi}}$ performs a more extensive analysis of the function and for `load` instructions that meet a set of conditions, instruments with a call to the `forwardFn` function. However, if the conditions are not met, it still instruments with the same `useFn` function.

Algorithm 4.3: Algorithm for normal Korat instrumentation

```
forall the instruction i in repOk do
    if i is a load instruction then
        insert call to useFn before i;
    end
end
```

There are four sets of conditions over load instructions that we instrument. Algorithm 4.4 gives the core algorithm using one condition for clarity, while the other conditions are described in Table 4.1. Algorithm 4.4 iterates over all `load` instructions in `repOk`. Each `load` instruction defines a new variable and provides a starting point for def-use analysis.

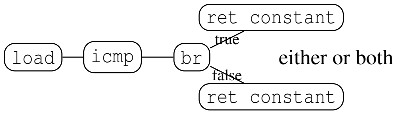
Algorithm 4.4: Algorithm for light-weight def-use analysis

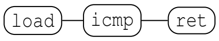
```
for every instruction i in repOk do
    if i is a load instruction then
        j = followUseChain(i);
        if j is an icmp instruction then
            k = followUseChain(j);
            if k is a br instruction then
                t = leadsToConstRet(true block of br);
                f = leadsToConstRet(false block of br);
                if t or f then
                    replace icmp with a call to forwardFn;
                    continue;
                end
            end
        end
        insert call to useFn before load;
    end
end
```

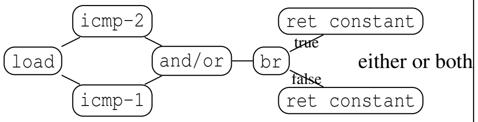
We perform simple def-use analysis on this variable traversing the `uses`

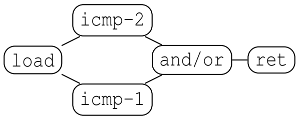
Table 4.1 Def-use analysis of `load` instructions we instrument.

No.	Use-chain	Description

1	 ret constant
	true
	load
	icmp
	br
	either or both
	false
	ret constant
		A `load` instruction defines a variable only used by an `icmp` instruction, the result of which is only used by a conditional `br` instruction leading to constant return on at least one of true and false sides.
		Example:
		`if(size!=visited.size())`
		`return false;`

2		A `load` instruction defines a variable only used by an `icmp` instruction, the result of which is used by a `ret` instruction. Example: `return size!=visited.size();`

3	 icmp-2
	ret constant
	true
	load
	and/or
	br
	either or both
	false
	icmp-1
	ret constant
		A `load` instruction defines a variable only used by two `icmp` instructions, where the results of both instructions are only used by an `and` or an `or` instruction, whose result is used by a conditional `br` instruction leading to constant return on at least one of true and false sides.
		Example:
		`if (data < min || data > max)`
		`return false;`

4		A `load` instruction defines a variable only used by two `icmp` instructions, where the results of both instructions are only used by an `and` or an `or` instruction, whose result is only used by a `ret` instruction. Example: `return data < min || data > max;`

list provided by LLVM and check if it is eventually (e.g. after sign-extending or bit-truncation) *only* used in a comparison instruction (`icmp`). This means that the accessed value is directly used as one argument of a comparison. We then follow the `uses` list of the result of the `icmp` instruction and see if it is eventually used *only* in a conditional branch instruction (`br`). This means that the comparison is used inside an `if` statement. Next, we inspect the true and false *basic blocks* (also called *then* and *else* basic blocks) that the branch leads to. Each basic block is a set of sequentially executed instructions ending with a terminating control flow instruction. If the terminating instruction is a return instruction (`ret`), we determine if a constant is returned. If a constant is returned in either the true basic block, or the false basic block, or both, we replace the comparison with a call to `forwardFn` with the original `icmp` operands as arguments to the function, along with boolean parameters determining which basic block lead to returning constant.

While inserting the call to `forwardFn`, we ensure that the true side of the comparison leads to the function returning `true` and the false side results in a `false` return. If not, we invert the comparison for passing it to `forwardFn` and again invert the return value from `forwardFn`. This simplifies the operation of `forwardFn`.

This high level description skims over two important details: (1) the details of following the `uses` lists and (2) determining if a basic block results in returning a constant.

Algorithm 4.5 describes the def-use analysis. To follow from one instruction to the next, we ensure that the target instruction is the only instruction that uses the result of the source instruction. If more than one instruction uses the result of the source instruction, we cannot determine if the other use does not influence the branch we will later take, and thus do not consider such a case. If the target instruction is a cast instruction (truncation, zero extending, or sign extending), we repeat the algorithm to find the instruction that uses the result of the cast. Cast instructions are common in LLVM because it is a typed language and there are no implicit type conversions.

Algorithm 4.5: Algorithm for following def-use chain.

```
input  : instruction i
if result of i has one use then
    j = only use of result of i if j is a cast instruction then
        return followUseChain(i);
    end
end
return null
```

Algorithm 4.6 gives the algorithm for determining if a basic block leads to a constant return. This works similar to constant propagation, except that the value to propagate (the result of comparison) is not really constant. Assuming that the result of comparison is known (`true` or `false`), we analyze if this result could have been propagated to the return instruction. Sometimes it can be propagated for a `true` result of the comparison, sometimes for a `false` result, and sometimes for both. This information is then used by `forwardFn` to mark candidates after performing a multi-value comparison.

The algorithm is used on both the true and false target basic blocks of a conditional branch instruction to determine if either side results in returning a constant. The function works by considering the last instruction in the basic block (the only control flow instruction). If it is an unconditional branch to another basic block, we recursively invoke the same function on the target of the unconditional branch. If, however, the terminating instruction is a return instruction (`ret`), we check the value returned. The returned value can be (1) a constant, (2) result of another instruction, or (3) a *phi constant*. A phi constant is a map from basic blocks to values

Algorithm 4.6: Algorithm to see if a block leads to a constant return.

```
input : BasicBlock b
i = last instruction in block b;
if i is ret instruction then
    v = value returned by i after resolving phi nodes;
    if v is constant then
        return v;

end
else if i is an unconditional branch then
    return leadsToConstantRet(target of i);
end
return null
```

where the value picked is based on the last basic block we were executing before a control flow instruction jumped into this block. The value corresponding to a basic block is again one of (1) a constant, (2) result of an instruction, or (3) another phi constant in the source basic block. Since we know the basic block chain from the first `load` instruction to this `ret` instruction, we recursively resolve the phi constants. It is possible that we are still unable to resolve it as it may depend on which basic block we came from before hitting the `load` instruction. When the returned value is a constant or a phi constant that we were able to resolve to a constant, we return this value.

Note that our def-use analysis is light-weight because we only attempt to find the *last* use of some field before every return. We also don't need to worry about another earlier `load` instruction accessing the same field. This would have been a consideration if we were making any decision statically. However, we delay our decision making until we actually execute the `repOk` predicate. At this time, we can easily monitor if a particular field is accessed for the first time or not (like standard Korat does) and use this information to enable forwarding for *that* access.

Table 4.1 shows all four conditions in which we instrument. Only the first condition is used in Algorithm 4.4 to describe the core concepts. The second condition is when the result of a comparison is directly returned without being used in a branch instruction. The next two cases are when two comparisons are made on the same field with an AND or an OR operation joining them. In such cases the variable defined by a `load` instruction is used in two `icmp` instructions and their results are used in an `and` or an `or` instruction whose result is used in a branch leading to a return (case 3) or directly returned (case 4). This can be easily generalized to multiple comparisons which are then joined by `and` or `or` instructions. However, our current implementations is limited to two comparisons.

Other limitations of our current implementation include instrumenting only one function (`repOk`). Any called helper functions are not instrumented. Additionally, we only support integer comparisons. These are, however, not fundamental limitations of the algorithm and more a matter of defining the scope of the implementation.

4.1.5 Correctness

For correctness, note that the candidates considered by Korat$_{\text{multi}}$ are the same as those considered by standard Korat. However, our candidates are divided into unmarked (complete `repOk` execution) and marked (identified during `forwardFn` as accepted or rejected without running `repOk` again) and the union of marked and unmarked candidates is the same as standard Korat. Thus, it suffices to show that marked candidates are correctly classified as accepted or rejected.

For marked candidates, we divide `repOk` into two parts. The first part goes from the start of `repOk` to the `forwardFn` call and the second part from the forwardFn call to the ret statement.

We determined using static data-flow analysis that a true or a false return from forwardFn would lead to a true or a false return from repOk. Thus we do not need to execute the second part once we know the return value from forwardFn.

On the other hand, for every marked candidate we *do* have an execution of the first part. That execution touched everything except the last field. This execution invoked the instrumented forwardFn and marked candidates based on the comparison of the last accessed field. This execution is shared by all candidates who only vary in this last accessed field. Thus we have a dynamic execution of the first part (shared by more than one candidate) and a static knowledge of the behavior of the second part. Thus, Korat$_{\text{multi}}$ generates the same set of valid inputs as the standard Korat algorithm.

4.1.6 Evaluation

We evaluate Korat$_{\text{multi}}$ on two metrics: the reduction in the number of repOk executions, and the time it takes to complete Korat analysis.

We use five complex structures to evaluate our technique. For each structure, we consider five different sizes. For each example, we generate structures of exactly the given size with unique elements. Our experiments were run on machines with two Intel Xeon 2.93GHz 6-core processors and 24GB of memory.

To instrument a structure whose definition (along with repOk and finitize methods) is given in struct.cc we use the following command:

```
    llvm-g++ --emit-llvm --no-exceptions -c struct.cc -o struct.o &&
llvm-ld -disable-inlining -disable-internalize struct.o korat.o -o korat &&
opt -load forward.so -forward struct.bc | opt -O3 | llc -o struct.S && g++
struct.S -o struct
```

The command works in the following stages: (1) translate user code in `struct.cc` to LLVM bit code, (2) combine the user code with the Korat algorithm in `korat.o`, (3) apply the LLVM analysis in shared library `forward.so` required by multi-value comparisons, (4) optimize instrumented code (inlining etc), (5) convert to native assembly, and finally (6) compile to native binary.

The structures we chose to test are min heap, dynamic order statistics, binary search tree, red-black tree, and sorted doubly linked list. Red-black trees are height balanced binary search trees using node colors and restrictions on assignment of that color. Dynamic order statistics are red-black trees where the nodes are further augmented with the size of the sub-tree rooted at them. The problem of order statistics is concerned with returning the k^{th} smallest number in a set. For example, the minimum element in a set of n elements is the first order statistic while the maximum is the n^{th} order statistic. Dynamic order statistics enable retrieving any order statistic in logarithmic time. All these structures form the basis of complex software. These structures have been used in evaluating other software analysis techniques including the original Korat algorithm [10].

The results of our experiments are given in Table 4.2. Our speedup ranges from 1.6X to 4.7X. This is pictured graphically in Figure 4.2.

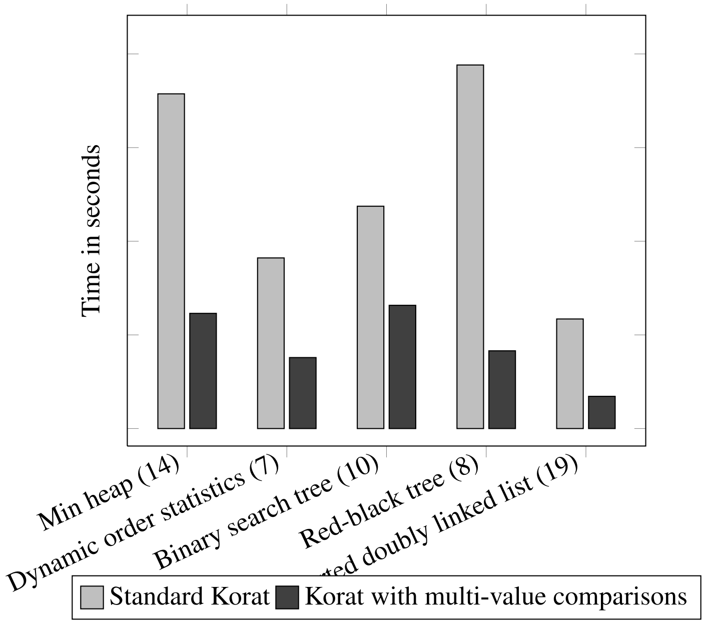
4.2 Focused Korat

Systematic black-box testing using constraints can often require generating and executing a large number of tests, which can be expensive. This section presents a novel technique to selectively reduce the number of test cases to be generated. Our technique applies across a class of structural constraint solvers. Experimental re-

Table 4.2 Comparison of Korat$_{\text{multi}}$ with standard Korat algorithm for structural constraint solving.

		Valid	Standard Korat		Korat with multi-value comparisons		
Subject	Size	Structures	Explored	Time [s]	Explored	Time [s]	Speedup
Min heap	9	896	64,401	0.14	19,781	0.04	3.5X
	10	3,360	316,369	0.85	94,538	0.26	3.3X
	12	79,200	9,277,511	34.74	2,961,691	11.50	3.0X
	13	506,880	55,005,301	4:04.25	18,545,942	1:23.24	2.9X
	14	2,745,600	356,649,476	29:47.13	120,077,299	10:15.61	2.9X
Dynamic order statistics	3	2	3,356	0.02	1,654	0.01	2.0X
	4	4	42,294	0.44	20,115	0.21	2.1X
	5	8	415,922	6.74	188,321	3.06	2.2X
	6	16	3,646,604	1:27.68	1,558,574	37.24	2.4X
	7	33	28,564,440	15:11.70	11,502,100	6:19.81	2.4X
Binary search tree	6	132	49,524	0.46	30,469	0.28	1.6X
	7	429	279,427	3.50	166,762	2.10	1.7X
	8	1,430	1,555,219	25.50	906,048	14.77	1.7X
	9	4,862	8,562,721	2:55.95	4,891,974	1:39.93	1.8X
	10	16,796	46,729,370	19:47.34	26,269,077	10:58.52	1.8X
Red-black tree	4	4	20,482	0.19	8,397	0.08	2.4X
	5	8	161,122	2.40	53,956	0.79	3.0X
	6	16	1,259,268	26.83	360,500	7.59	3.6X
	7	33	7,962,572	3:52.97	1,938,263	55.66	4.2X
	8	56	51,242,194	32:21.49	11,077,150	6:55.75	4.7X
Sorted doubly linked list	15	1	1015748	19.20	294,897	5.88	3.3X
	16	1	2,162,624	44.75	622,576	13.83	3.2X
	17	1	4,587,452	1:49.12	1,310,703	32.65	3.3X
	18	1	9,699,256	4:12.08	2,752,494	1:14.80	3.4X
	19	1	20,447,156	9:45.19	5,767,149	2:52.19	3.4X
						Average speedup	2.9X

Figure 4.2 Time taken by Korat with and without multi-value comparisons. Number is parenthesis is the size of structures generated.



sults show that the technique enables an order of magnitude reduction in the number of test cases to be considered.

We call this technique *focused generation* [98] and develop it for Korat. Our technique addresses the concern of "what to generate" in a constraint solver.

Focused generation means bounded exhaustive test generation where the exhaustive nature of test generation is *focused* on specific fields. For every possible assignment for the fields which are focused, the generation should find only one valid assignment for the remaining fields. This allows optimizing bounded exhaustive testing when objects are composed of objects of different classes that have been tested before. To illustrate, consider the binary search tree discussed above with some library class (e.g. Set) as data instead of integer data. Baseline Korat would generate the entire structure at the concrete level (including exhaustively testing the inner Set class). Focused generation allows more efficient testing of the subject class assuming the correctness of the classes it depends on. The knowledge of an abstraction function suffices for an effective application of this optimization in this scenario.

Focused generation also allows focusing generation towards some aspect of a data structure. For example in our tests we focus generation on the structure of a sorted singly linked list. Thus for a given node structure we get one sorted assignment of numbers however big the allowed range of numbers is. Without focused generation we will get a different test case for every possible sorted subsequence of the given range.

Since focused generation produces a smaller set of results, it cannot be strictly called an optimization of Korat algorithm. However, Korat provides no way of bounded exhaustive testing (BET) for a structure without doing BET for sub-structures. It also provides no way of doing BET for one aspect of the structure but not others. In such cases, we need to filter the results to reduce test suite size and retain the desired tests. Focused generation is an optimization for this procedure of applying Korat and filtering results.

To illustrate how to focus generation on an aspect of a structure, consider the same example of an ordered binary search tree. If we want to test some function that operates on structural aspect of the binary search tree, we may want to generate one binary search tree of each shape with valid data assignments. However, Korat will try to generate all possible data assignments for each valid tree shape. Focused generation can prune out these multiple assignments, leaving a single valid assignment for data. This results in fewer explored states and faster generation. For focused generation to work, the user marks the fields out of focus in the finitization. By default, all fields are in focus, and that means the same behavior as standard

Korat. For ordered binary search trees up to three nodes, standard Korat finds 15 valid candidates after exploring 178 candidates. All the trees of size 1 and 2 are repeated thrice with different data values. Focused generation only produces nine candidates; each of the trees in Figure 2.1 with one valid assignment. These nine trees are found after exploring 127 candidates. Thus an additional 51 candidates are pruned out.

4.2.1 Implementation

Korat with focused generation is shown in Algorithm 4.7. This modified Korat works on unmodified `repOk` constraints. The function ValidCandidate referred in Algorithm 4.7 can take any action on valid candidates, like count them, store them, or directly test some code using them.

To implement focused generation, we introduce a `onesol` array and a `lastvalid` boolean. On line 1 in Algorithm 4.7 the variable `lastvalid` is initialized to *false* and on line 5 it is assigned the result of last candidate checked. The `onesol` array is a boolean array containing one entry for each field. To enable this optimization, the user has to mark the field as requiring just one solution (out of focus). This is done using an additional parameter when adding the field in the finitization. This additional parameter results in the corresponding index in `onesol` array to become *true*. The check on line 10 in Algorithm 4.7 sees if the last candidate was valid and the user wanted only one solution for the current field, then skip over all remaining values of this field and move to the field accessed before it. If the previous field also needed just one solution, its values are also skipped and we move on. Thus significant portions of state space are pruned out and a lot fewer test cases are generated. All this requires no change in the predicate itself.

Algorithm 4.7: Korat with focused generation. A new array *onesol* remembers which fields need only one solution. Global *lastvalid* remembers if the last candidate checked was valid or not.

```
input : candidateV
candidate $\leftarrow$ BuildCandidate(candidateV);
(predicate, accessFields, masked) $\leftarrow$ repOk(candidate);
if predicate then
    ValidCandidate(candidate);
end
for $i \in$ accessFields do
    pruning[$i$] $\leftarrow$ false;
end
while Size(accessFields)>0 $\wedge$ candidateV[Top(accessFields)]=0 do
    for $i \leftarrow$ 1, NonIsoMax(candidateV, accessFields) do
        if not masked[Top(accessFields), i] then
            candidateV[Top(accessFields)] $\leftarrow i$;
            Korat(candidateV);
        end
    end
    for $i \leftarrow$0, MaxDomainIndex(Top(accessFields)) do
        masked[Top(accessFields) $\leftarrow$ false;
    end
    candidateV[Top(accessFields)] $\leftarrow$ 0;
    pruning[Top(accessFields)] $\leftarrow$ true;
    Pop(accessFields);
end
```

Table 4.3 Results of running Korat with and without focused generation on Sorted Singly Linked List and Red-Black Trees. For each structure, 4 different sizes are tried. Timeout is set to 10 minutes.

		Without Focussed Generation	With Focussed Generation
Subject	Size	Found / Explored	Found / Explored
Sorted Singly Linked List	20	1048575 / 179307145	20 / 1770
	25	TIMEOUT	25 / 3275
	100	TIMEOUT	100 / 176850
	200	TIMEOUT	200 / 1373700
Red Black Tree	10	14101 / 4901265	377 / 1903225
	11	40074 / 20130233	707 / 7301092
	12	112813 / 85201572	1395 / 28239078
	13	TIMEOUT	2835 / 109591533

4.2.2 Evaluation

To evaluate focused generation, we have implemented it in Korat for C++. Our implementation can enable or disable focused generation. With it disabled, we get behavior identical to standard Korat. We compared outputs with the Java implementation to ascertain correctness. The experiments were run on a Pentium 2.8GHz processor with 4GB memory running Linux.

To see the effect of focused generation, we take a sorted singly linked list and a red-black tree. We focus testing on structural correctness and not on data ordering requirement. Thus we find all structures with one satisfying data assignment. Results of this experiment are given in Table 4.3.

For both these structures, we took Standard Korat and Korat with Focused Generation. We test four sizes of each structure on both versions of Korat. We compare how focused generation improves Korat in terms of time and in terms of reducing valid candidates generated.

We observe that focused generation, where applicable, can improve time drastically in some cases. For sorted singly linked list, the benefit is enormous because we can avoid all sorted sub-sequences of data. For red-black trees, focused generation takes about three times less time.

The other observation is that because focused generation is generating fewer candidates, the actual testing time will reduce. For example test execution phase for red-black tree of up to size 14 will run 154 times faster in case of focused generation. This is assuming that all generated tests are used for testing.

4.3 Parallel Korat

This section presents PKorat [95], a new parallel algorithm that parallelizes the Korat search. PKorat explores the same state space as Korat but considers several candidates in each iteration. These candidates are distributed among parallel workers resulting in an efficient parallel version of Korat. Experimental results using complex structural constraints from a variety of subject programs show significant speedups over the traditional Korat search.

For efficient search, Korat uses a dynamic analysis of the predicate: Korat search prunes large portions of the input space by monitoring predicate executions on candidate inputs. While the dynamic analysis enables efficient pruning, it renders the Korat search inherently sequential: without executing the predicate on a candidate input, Korat cannot determine the next candidate input. Moreover, a simple partitioning of the state space and a distributed search does not yield high speed-ups since it is not possible to predict which parts of the state space would be pruned and exploring those parts is simply redundant.

Our key insight is that even though it is not feasible to *fast-forward* Korat search effectively, the systematic exploration of the input space can still be parallelized by exploring non-deterministic field assignments in parallel. PKorat search backtracks by generating *several* candidate inputs that are explored in parallel. A key aspect of PKorat is that it explores exactly the same number of candidates that Korat does and still enables a scalable parallel implementation. PKorat uses a master-slave configuration to implement its parallel search. When delegating exploration to a slave processor, PKorat provides a small amount of meta-information that prevents different slaves from exploring the same candidates and avoids redundant exploration. Misailovic et al. [77] provided techniques that efficiently parallelize test execution, while test generation was mostly sequential (Section 7.3.1).

4.3.1 Illustrative Example

Korat produces only one candidate vector after testing the previous one. PKorat, on the other hand, produces a list of candidate vectors after testing each candidate. Furthermore, all these candidates are produced by Korat as well. The working of PKorat on the same example as in Section 2.2.1 is given below. The equivalence of candidates explored by Korat and PKorat is discussed in Section 4.3.2.4.

Consider that PKorat is working on the candidate given in Section 2.2.1 and the ordered list of accessed fields during `repOk` is $\langle root, N_0.left, N_0.right, N_1.left, N_1.right, size \rangle$.

PKorat, like Korat, finds the maximum value for `size` which in this example is 3. As it already maximum, it is reset to minimum (which is also 3) and maximum value for $N_1.right$ is calculated and found to be N_2. At this point, PKorat differs in that it generates three candidate vectors with the value of $N_1.right$ set to N_0, N_1, and N_2, respectively. It then finds the maximum value of $N_1.left$ and produces three more candidates with value of $N_1.left$ set to N_0, N_1, and N_2, respectively. Another three vectors are produce by setting $N_0.right$ to N_0, N_1, and N_2, respectively. When we observe that $N_0.left$ is not at index 0 we have to stop since the current vector would have been produced in parallel by a similar previous step which catered $N_0.left$ and anything before it. After producing all these nine candidate we can distribute them and test them with a predicate in parallel. Each distributed task consists of evaluating a candidate and distributing the subsequent candidates again.

The motivation behind producing all these candidates is based on the fact that Korat cannot prune a candidate all of whose non-zero fields are accessed. Using the ordered list of accessed fields, we make a candidate by changing some field while keeping everything accessed before this field unchanged, and everything accessed after it, pointing to the first value in their domain. Every non-zero field in this candidate would be accessed, because we assume `repOk` is deterministic and every field accessed before the changed field has the same value. Therefore Korat does not prune it. Furthermore stopping at first non-zero accessed fields protects against the same candidate produced out of two different candidates. Both of these facts are formally proved in Section 4.3.2.4.

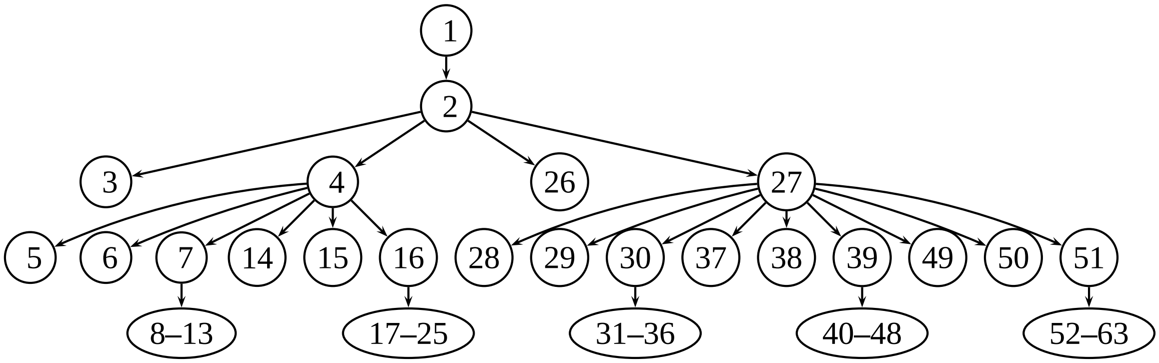
Figure 4.3 shows how the 63 candidates explored by Korat are explored by PKorat. The numbers represent the order in which Korat explores them sequentially. Note that the example candidate 27 produces nine candidates as discussed above.

Given enough processors and ignoring overheads, the parallel algorithm can complete test generation in 5 units of time instead of 63 units. A quad-core can ideally complete the task in 18 units of time giving 3.5X speedup. The actual speedup is lower due to communication overheads.

4.3.2 Algorithm

PKorat uses a master slave configuration [68], where processor 0 is always designated as the master processor. It contains a work queue of candidate vectors. Every other processor acts as slave and contacts the master for a candidate vector

Figure 4.3 Tree of explored candidates showing Parallel Korat search progress, where the numbers represent the sequential order of their exploration in standard Korat (last level nodes aggregated). Note that given enough processors and ignoring overheads, PKorat can ideally explore this state space in 5 units of time, whereas Korat would take 63. Even a quad-core can complete in 18 units of time giving 3.5X speedup.



to process. After processing it send a list of candidate vectors back to the master, and the master enqueues them.

4.3.2.1 Master Algorithm

Algorithm for master process is given in Algorithm 4.8. The master processor maintains a list of free slaves (`slaveQ`) and a list of pending candidate vectors (`workQ`). It initializes work queue with an all zero candidate vector (`ZeroV`). It then loops while receiving and processing slave requests.

There are two types of requests. A QUEUE request and a DEQUEUE request. The former is used to add candidate vectors to the work queue, while the later is used to fetch a candidate vector. If a candidate vector is not readily available, the slave is remembered in slave queue. Whenever candidate vectors become available, they are sent to these free slaves instead of being queued in work queue. The algorithm terminates when every slave is in slave queue. At this stage the work queue

Algorithm 4.8: Algorithm for master processor in PKorat. It keeps a queue of waiting slaves and a queue of candidate vectors. On a dequeue request, a candidate vector is sent if available, otherwise the slave goes in waiting queue. On a queue request, the candidate vector is sent to waiting slaves, or stored in candidate queue if none are waiting. The algorithm terminates when every slave is in waiting queue.

```
workQ $\leftarrow$ createQueue();
slaveQ $\leftarrow$ createQueue();
Queue(workQ, ZeroV);
while Size(slaveQ) $\neq$ SlaveCount do
    (slave, cmd, candidateV) $\leftarrow$ Recv(any);
    if cmd = Queue then
        if Empty(slaveQ) then
            Queue(workQ, candidateV);
        else
            slave$'$ $\leftarrow$ Dequeue(slaveQ);
            Send(slave$'$, Work, candidateV);
        end
    else if cmd = Dequeue then
        if Empty(workQ) then
            Queue(slaveQ, slave);
        else
            candidateV $\leftarrow$ Dequeue(workQ);
            Send(slave, Work, candidateV);
        end
    end
end
forall the s $\leftarrow$ slaveQ do
    Send(s, Exit);
end
```

must also be empty. As soon as it happens, it sends all slaves an EXIT message.

4.3.2.2 Slave Algorithm

Slave processors are responsible for processing a given candidate vector and producing zero or more candidate vectors to be processed next. Whereas the master processor is only concerned with seamless communication, novel contributions of this dissertation manifest in slave processing. The algorithm is given in Algorithm 4.9.

Every slave sends the master processor a DEQUEUE request, receives a candidate vector (`candV`), processes it, possibly sends back a QUEUE request, and then repeats the whole thing. It terminates when a DEQUEUE request results in an EXIT response from the master.

To process a candidate vector, the slave first converts the candidate vector to an actual candidate C++ data structure using BUILDCANDIDATE. This function not given here, is identical to the one used in Korat. It works by assigning all fields of all objects involved according to domain indices stored in the candidate vector. Section 2.2.1 shows the assignments for the example discussed in Section 2.2.1 and Section 4.3.1.

The predicate REPOK is then used to validate the given candidate. VALIDCANDIDATE is invoked if this candidate is valid. This function, also not given here, is a placeholder for *any* processing of valid candidates. For example, it can count the number of valid candidates, store the candidate in a file for later use, or even invoke the actual program for which these tests are generated. The last two options are similar in nature to offline and online test execution in previous work on parallelizing Korat [77]. However, as discussed before, our solution does not share its limitations.

Algorithm 4.9: Algorithm for slave processors in PKorat. It builds a C++ object structure using BuildCandidate. Then tests it using repOk. Then for all accessed fields up to a field pointing to a non-zero index, it generates candidates for all non-zero indices of that field up to the maximum index given by NonIsoMax. NonIsoMax is given in Algorithm 4.10

```
Send(master, DEQUEUE);
(cmd, candV) $\leftarrow$ Recv(master);
while cmd = WORK do
    candidate $\leftarrow$ BuildCandidate(candV);
    (pred, accV) $\leftarrow$ repOk(candidate);
    if pred then
        ValidCandidate(candidate);
    end
    while $Size(accV)>0 \wedge Top(accV)=0$ do
        field $\leftarrow$ POP(accV);
        for $i \leftarrow 1$, NonIsoMax(candV, accV, field) do
            candV[field] $\leftarrow i$;
            SEND(master, QUEUE, candV);
        end
        candV[field] $\leftarrow$ 0;
    end
    SEND(master, DEQUEUE);
    (cmd, candV) $\leftarrow$ RECV(master);
end
```

Lastly, candidate vectors to be explored next are found. The algorithm works using the accessed fields stack (accV) returned by REPOK. This contains all fields accessed by REPOK in making its decision, in the order of first access. The slave pops fields one by one off this stack until a non-zero field is found. For each popped field, it produces candidate vectors for all non-zero valid values of this field. The maximum valid value is given by NONISOMAX and is discussed in Section 4.3.2.3. It stops at a non-zero field to avoid producing the same candidates as some other slave would do. This is discussed and proved in Section 4.3.2.4.

Actual implementation aggregates candidate vectors to be sent back to master in fewer messages. It also uses double buffering at client so that a candidate vector can be received and ready to be processed while the previous one is being processed. These are considered as implementation optimizations and excluded from the algorithm description here, to emphasize key contributions of this work.

4.3.2.3 Non-isomorphism

Isomorphic candidates are candidates that only differ in the identities of their objects. By swapping all members of any two objects and then swapping all references to these objects, an isomorphic copy of the original structure can be formed. For example, isomorphic copies of trees in Figure 2.1 can be formed by using N_1 as the root, and N_0 in place of N_1. Most programs do not make decisions based on object identities (e.g. their memory address) and therefore testing isomorphic copies is a waste of time.

Our algorithm for avoiding isomorphic copies is identical to Korat [10]. A simplified version assuming that NULL is always tested for object fields and assuming no polymorphic fields (fields that take objects of more than one type) is given in Algorithm 4.10. For non-primitive fields the algorithm checks all previously accessed fields having the same domain, and finds the maximum index accessed (`touched`). If any untouched objects are left, it allows one of them to be tried.

Algorithm 4.10: Helper function for slave processors in PKorat. It returns the maximum valid index for primitive fields. For non-primitive fields it ensures that no more than one untouched object is tried. An object is touched if a previously accessed field with the same domain points to it.

```
input  : candV, accV, field
$m \leftarrow$ MaxDomainIndex(field) ;
if NonPrimitive(field) then
    touched $\leftarrow$ 0;;
    forall the $i \in accessedV$ do
        if SameDomain(i, field) then
            touched $\leftarrow$ Max(candV[$i$], candV[field]);
        end
    end
    m $\leftarrow$ Min(m, touched+1)
end
return $m$
```

Even though the basic algorithm for isomorphism avoidance is identical to Korat [10], there is an inherent efficiency in PKorat. Korat needs to repeat this procedure to find the valid maximum in every iteration. It optimizes this by using caching, so that the maximum for a given field, with the same values of fields accessed before it, is calculated only once. Our parallel algorithm however generates all such candidate vectors at the same time. Therefore it only calculates the valid maximum once for each field with the same values accessed before it and no caching is needed either.

4.3.2.4 Completeness and Soundness

We prove completeness and soundness of PKorat by proving equivalence with Korat for a deterministic `repOk` function. Completeness and soundness of Korat has been proven elsewhere [10]. We prove equivalence by showing that PKorat explores a candidate if and *only* if Korat explores it.

Proof. PKorat pops fields from accessed field stack and stops at the first field set to a non-zero index. This is because PKorat tries all non-zero indices at the same time. Thus, the time this field was set to the current non-zero value, all other mutations would have been tried as well. Similarly, mutation of previous fields would have either been produced at the same time or would have been produced at some previous invocation by applying the argument recursively. Therefore PKorat produces the immediate next candidate (according to Korat sequence) either at the same time or has produced it previously. Therefore every candidate produced by Korat is produced by PKorat as well. This proves the *if* direction.

A candidate is pruned out by Korat only if there is another explored candidate with the same values for all fields accessed by `repOk`. This explored candidate has all non-accessed fields set to zero index. This is because Korat generates a new candidate by mutating only the last accessed field. For a deterministic `repOk`, this implies that *exact* same fields will be accessed again and possibly some more. When Korat backtracks, fields up to the backtracked field must be accessed and possibly some more. Thus all fields that are not certain to be accessed are at zero index. Therefore, if all non-accessed fields of a candidate are at zero index, it cannot be pruned out by Korat. Since PKorat only produces such candidates, we can infer that they are produced by Korat as well. This proves the *only if* direction and completes our proof. □

4.3.3 Evaluation

To evaluate the efficiency of PKorat, we implemented it in C++ using Message Passing Interface (MPI) [100] library. We also implemented the serial Korat algorithm in C++ so that it can be run on the same machines and meaningful performance comparison can be made. We have run the experiments on a Linux cluster provided by Texas Advanced Computing Center (TACC). The cluster consists of 1,300 nodes, with 2 dual core processors per node, for a total of 5,200 cores. The serial tests were run on a single core of this machine while the parallel runs used more cores.

Our test subjects include standard data structures such as a singly linked list, a binary tree (as given in Section 4.3.1), and a red-black tree. We also test directed acyclic graphs (DAG) to compare performance with previous work and to discuss issues with isomorphic graphs. We then give a novel application for automated generation of all valid Java class hierarchies with a given number of classes and interfaces.

We run our experiments with various number of processors to show the scalability of our algorithm. For each experimental run, we give the speedup from serial version and the clock time for the execution. We discuss specific details and experimental observations in the following Sections.

4.3.3.1 Singly Linked Lists, Binary Trees, and Red-black trees

We test PKorat and compare it to Korat for singly linked lists, binary trees, and red-black trees. These standard structures have been commonly used to evaluate the performance of algorithms for generating structurally complex test inputs. This experiment shows the benefits of using parallel processing for generating these structures.

Table 4.4 Results of Korat and PKorat for a number of different data structures. For each parallel run, there is one master and the rest are slaves. When the subject data structure has many nodes, the parallel versions give the most benefit, up to 49.5X on 64 nodes. Quad-core performance in all cases is exceptional, considering that only three processors are actually working on candidates it is between 1.9–2.6X.

Subject	Proc	Time (s)	Speedup
Singly Linked List (500 nodes)	Serial	1387	1.0X
	4p	523	2.6X
	16p	108	12.8X
	64p	28	49.5X
Binary Tree (13 nodes)	Serial	480	1.0X
	4p	250	1.9X
	8p	120	4.0X
	16p	105	4.6X
Red Black Tree (10 nodes)	Serial	347	1.0X
	4p	136	2.6X
	8p	82	4.2X
	16p	77	4.5X
Directed Acyclic Graph (7 nodes)	Serial	1163	1.0X
	4p	544	2.1X
	8p	318	3.7X
	16p	284	4.1X
Java Class Hierarchies (8 nodes)	Serial	149	1.0X
	4p	71	2.1X
	8p	45	3.3X
	16p	39	3.8X

For singly linked lists, we were able to explore 500 node lists in a reasonable time. Using PKorat for such big structures proved the most beneficial. We were able to get 49.5X speedup with 64 processors. Due to complexity of state space search for binary tree and red-black tree, Korat is able to search 13 node for the former and 10 nodes for the later within a 30 minute limited we posed. They showed scalability up to 16 processors. If we had more nodes like singly linked lists, the speedup would be better. However the clock time to run the experiment would also be much more.

Note that communication costs rise with more nodes. That is because when processors are allocated from a big cluster, they are not always close together and there is a single master to communicate to. Therefore, even though PKorat cannot deteriorate in performance with more processors, it does deteriorate in practice after an optimal number of processors due to increased average node to node communication time.

4.3.3.2 Directed Acyclic Graphs (DAG)

Directed Acyclic Graphs (DAG) can be represented as shown below.

```
class DAG {
  class Node {
    std::vector<Node*> children;
  };
  std::vector<Node*> nodes;
};
```

To verify the acyclicity constraint, we can do a depth first search from each starting node and find if any node can be reached twice. This scheme, however, leads to a number of equivalent inputs, i.e. structures that are different at the concrete level but represent the same DAG. Detailed discussion of this topic can be found in previous work [77], which also discusses an optimized `repOk`. This optimized `repOk` is more restrictive and would even reject some valid graphs. The end result is fewer generated graphs without missing any unique graph. The algorithm is based on descendant counting and ordering.

We use a very simple and less compute intensive approach that results in even fewer generated graphs. We write a `repOk` that only accepts topologically sorted graphs. Furthermore, nodes in any array are ordered in increasing order of some identifier. It is easy to prove that this approach generates all graphs since every DAG can be topologically sorted. But equivalent graphs are still produced as topological sorting of a DAG is not unique. For a DAG of six nodes, a `repOk` that accepts every valid graph explores 16,216,503 structures and declares 1,336,729 as valid. The optimized `repOk` [77] explores 2,628,140 structures with 21,430 valid instances. Our simple topological sort based `repOk` explores merely 517,743 structures but validates 32,768 of them. Thus generation is much faster but testing would be slower due to more test cases generated than previous work. For comparison, note that the actual number of non-isomorphic graphs of size 6 is only 5,984 [77].

Results show that we can more than double the speed on a quad-core. Note that only three processors are actually evaluating candidates on a quad-core, so the speedup on 2.1 of a maximum possible of 3 is actually significant. Performance keeps on improving for more processors until speedup crosses 4. After that, enough parallelism is not generated to exploit additional processors. Rise in average communication costs actually decreases speedup for more processors.

4.3.3.3 Java Class Hierarchies

We use Korat to generate all valid Java programs with a given number of classes and interfaces. We then show the performance gains with our parallel algorithm. Automated generation of valid programs can be very useful in testing of compilers and associated tools. We define the three classes for this experiment.

```
class JavaClass {
  JavaClass* extends;
  std::vector<JavaInterface*> implements;
};
class JavaInterface {
  std::vector<JavaInterface*> extends;
};
class JavaProgram {
  std::vector<JavaClass*> classes;
  std::vector<JavaInterface*> interfaces;
};
```

A JavaProgram class whose lone instance represents the program to be generated. JavaClass and JavaInterface classes representing classes and interfaces, respectively. Finitization puts bounds on the number of classes and interfaces. The `repOk` function checks that the classes form a valid class hierarchy. One of the results produced is shown here.

```
public interface I1 {}
public interface I2 {}
public class C1 {}
public class C2 extends C1 implements I1, I2 {}
```

This problem is similar to directed acyclic graphs with some additional checks. These additional checks allow exploring a slightly larger state space in less time. We can see that quad-core performance is still similar at more than twice the speed. While the maximum speedup attained is around 4. For more nodes, the maximum speedup is more, and less for less nodes. Thus the bigger the problem space, the bigger the potential speedup.

Chapter 5

Constraint-driven staged testing

This chapter describes the Pikse suite of techniques for improving constraint-driven analysis for combined black-box and white-box testing. We develop our techniques in the context of the Korat algorithm and symbolic execution. Specifically, we consider the problem of testing programs with preconditions (Section 5.1) and introduce *staged* symbolic execution, where such programs are tested in stages using a combination of black-box and white-box techniques. Staged symbolic execution was initially presented at SAC 2012 [96].

5.1 Testing in the presence of pre-conditions

The problem of test generation using pre-conditions has been studied in various research projects over the last decade [10, 37, 64, 76, 91] using two primary approaches: black-box testing and white-box testing. In black-box testing [10, 37, 76], the pre-condition is taken in isolation of the method under test and used to enumerate valid concrete inputs, which are later used to test the method. The key advantage of this approach is its ability to generate *dense suites*, which allow bounded exhaustive testing, which has been shown to be effective at finding bugs in a variety of applications, including compilers [38], refactoring engines [27], service location architectures [76], and fault-tree analyzers [106]. A basic limitation of this approach however is the need to generate a large number of tests and the need to run each of those tests against all methods under test — even if for certain methods, many tests are equivalent.

In contrast, in white-box testing, the program "`if (repOk()) m();`" is directly used for test generation, e.g., using symbolic execution [64, 91]. The key advantage of this approach is its ability to directly explore a large number of paths in the method under test and to generate test suites that achieve high code coverage and likely contain fewer tests than black-box approaches. A basic limitation of this approach however is the need to repeatedly consider symbolic execution of the *repOk* method for each method under test – by construction, symbolic execution must execute "`if (repOk()) m();`" for each *m* that has *repOk* as its pre-condition. Thus, this approach requires enumeration of valid inputs from scratch for each method under test — even if certain methods have the same pre-condition. While re-use of concrete inputs generated for one method, say *m*, to test another method, say m', is possible, this white-box approach for method *m* then degenerates into a black-box approach for method m'.

We present a novel approach [96] to increase the efficiency of symbolic execution for systematic testing of object-oriented programs. Our insight is that we can apply symbolic execution in *stages*, rather than the traditional approach of applying it all at once, to compute *abstract symbolic inputs* that can later be shared across different methods to test them systematically. For example, a class invariant can provide the basis of generating abstract symbolic tests that are then used to symbolically execute several methods that require their inputs to satisfy the invariant. We present an experimental evaluation to compare our approach against KLEE, a state-of-the-art implementation of symbolic execution. Results show that our approach enables significant savings in the cost of systematic testing using symbolic execution.

We specifically consider the problem of how to use symbolic execution to systematically enumerate inputs for a C++ method *m* that has a pre-condition *p*, which is also written in C++ as a *repOk* predicate, i.e., a method that inspects its inputs to check the pre-condition and returns true if and only if it is satisfied. Our goal is to provide efficient and effective enumeration of *valid* inputs for *m*, i.e., inputs that satisfy the pre-condition. The key technical challenge in solving this problem is to lead symbolic execution into the body of *m*, which must be preceded by an invocation of *repOk*, since correct behavior of *m* requires the pre-condition to hold. Thus, symbolic execution must be performed on the program "`if (repOk()) m();`".

We apply symbolic execution in stages. The first stage performs symbolic execution of *repOk* to generate abstract symbolic tests, which are object graphs that have symbolic components defined by constraints, akin to path conditions in symbolic execution. Thus, one abstract symbolic test represents (possibly) many concrete tests, and the suite of abstract symbolic tests compactly represents a likely much larger suite of concrete tests. The second stage takes, in turn, each method under test that has the same pre-condition (as defined by *repOk*), and symbolically executes it using each abstract symbolic test.

Symbolic execution during the second stage dynamically expands each abstract symbolic test into a number of concrete tests based on the control-flow of the method under test. Methods that require more tests due to complex control-flow are tested using more tests and methods that require fewer tests are tested against fewer tests. While this control-flow-driven exploration during the second stage allows our approach to share benefits of white-box techniques, the use of specifications in the first stage enables our approach to share a key benefit of black-box techniques: generation of abstract symbolic tests can proceed even before the code to test is implemented, much in the spirit of test-first programming — this contrasts with other approaches based on symbolic execution, which a priori require an implementation of code under test. We believe our approach provides a sweet-spot for using symbolic execution for systematic testing of methods with pre-conditions.

A major advantage of our approach is the re-use of abstract symbolic tests across different methods. The re-use further increases in the context of software evolution. Abstract symbolic tests are generated with respect to specifications. If some specifications remain unchanged while code evolves, the same abstract symbolic tests can be re-used for the new version of code. Thus, our approach holds potential for significant savings in regression testing effort.

Our work opens a novel avenue for tackling the problem of scaling symbolic execution. Our experimental evaluation demonstrates the potential of our approach in the context of well-studied subject programs. However, abstract symbolic tests are not limited to object graphs with constrained symbolic components. To illustrate, an abstract symbolic test may represent a partial XML document or a partial Java program with symbolic elements that have constraints, thereby enabling a novel approach for testing of systems that take XML or Java programs as inputs, such as compilers and refactoring engines. We believe staged symbolic execution and abstract symbolic tests provide precisely the technical elements that are needed to take approaches based on symbolic execution, such as dynamic symbolic execution [16, 42, 109], aka concolic execution [91], to a higher level of efficiency and effectiveness.

5.2 Motivational example

To exhaustively test the `add` method of binary search tree for three nodes using a black-box approach, we need to store 53 tests and run $53 \times 4 = 212$ tests (there are 4 distinct values that can be added to the tree). To test the same method using symbolic execution, we do not need to provide a bound on any field.

For the add method, we write the following code and run it through symbolic execution.

```
SearchTree b;
if (b.repOk()) { // pre-conditions
  try {
    b.add (x); // x is a symbolic integer
    // check b using post-conditions
  } catch(...) {
    // report implementation bug
  }
}
```

Based on the symbolic execution interpreter used, we may also need to add code to explicitly mark the fields as symbolic. For example, for lazy initialization [64], we write getters functions for each reference field that lazily initialize a field from its field domain at first access.

The essence of this technique is that test generation and test execution are combined as one. Symbolic execution of repOk prunes invalid choices and continues to the method under test for valid choices. If the tests are concretized after repOk, symbolic execution will provide one valid instance for each object graph. However, it may not cover all paths in the method under test as symbolic execution was unaware of the path condition(s) that *will* be formed during the execution of the method under test.

This provides our motivation for a technique that can store abstract tests with a symbolic state. This symbolic state should be restored when the test is executed and non-reference fields can be concretized as needed during symbolic execution of the method under test. Such a technique would combine the benefits of both white box and black box testing for methods with structurally complex inputs.

Figure 5.1 `repOk` symbolically executed as part of every method tested with symbolic execution.

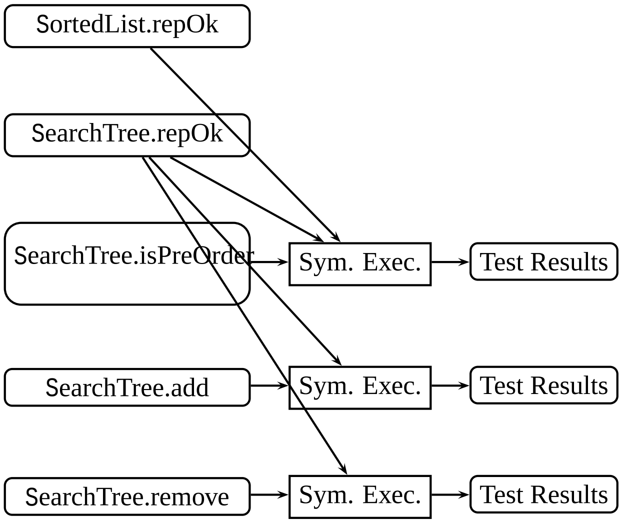
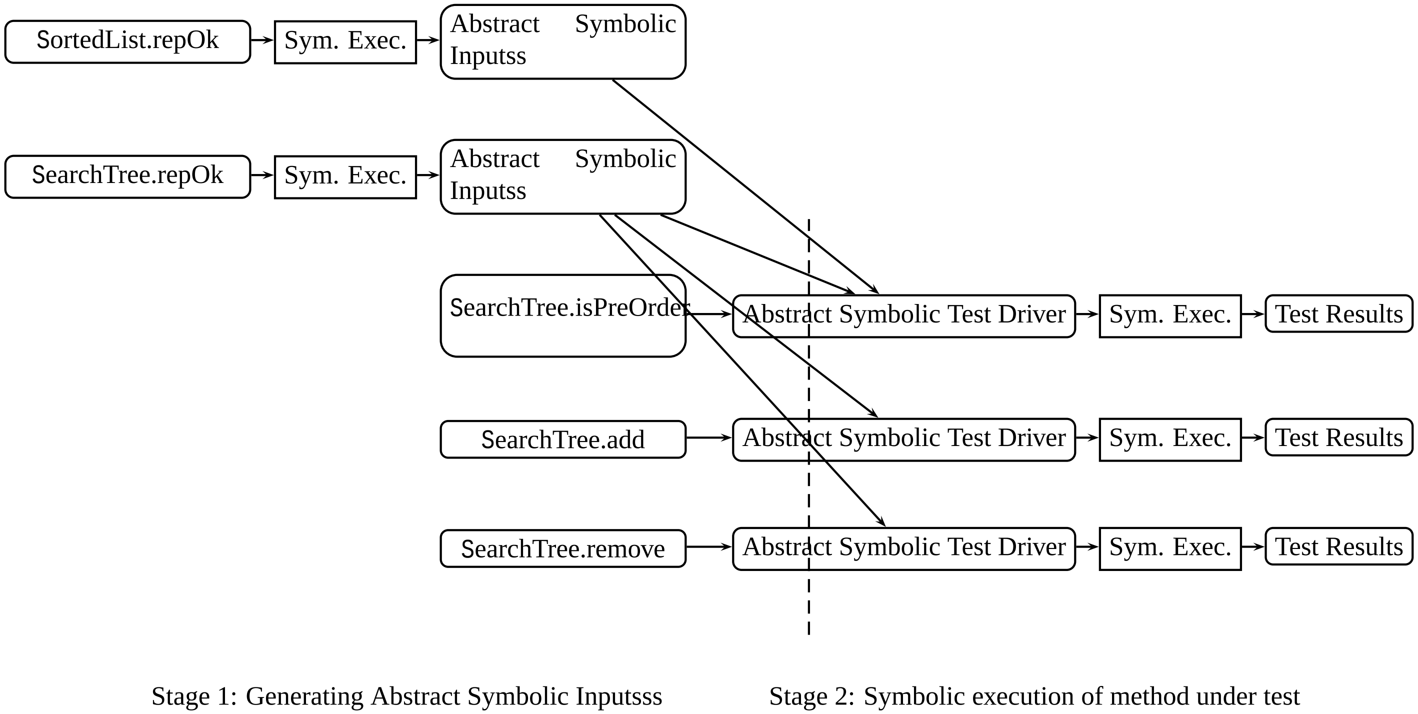


Figure 5.2 Staged symbolic execution of `add`, `remove`, and `isPreOrder` functions of binary search tree.


SortedList.repOk
Sym. Exec.
Abstract Symbolic Inputss
SearchTree.repOk
Sym. Exec.
Abstract Symbolic Inputss
SearchTree.isPreOrder
Abstract Symbolic Test Driver
Sym. Exec.
Test Results
SearchTree.add
Abstract Symbolic Test Driver
Sym. Exec.
Test Results
SearchTree.remove
Abstract Symbolic Test Driver
Sym. Exec.
Test Results
Stage 1: Generating Abstract Symbolic Inputsss
Stage 2: Symbolic execution of method under test

5.3 Overview & High-level architecture

The high-level architecture of staged symbolic execution is explained with three example methods. We consider the `add`, `remove`, and `isPreOrder` methods of a binary search tree. The first two add and remove an element respectively while the third takes a sorted linked list and checks if it contains the pre-order traversal of the binary search tree. In Figure 5.1, the high level steps involved in their symbolic execution are shown. The symbolic execution engine takes the class invariants for the concerned object(s) *and* the method under test and explores the *complete* search space. The class invariant is symbolically executed whenever a method needs a valid object of that type.

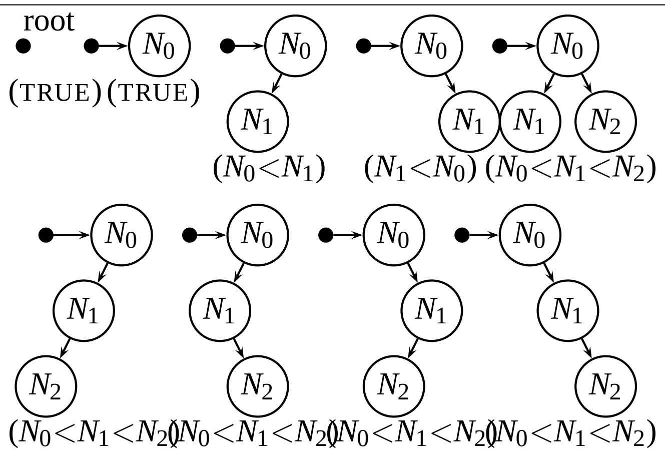
Staged symbolic execution differs in that symbolic execution of one class invariant is only done once. Figure 5.2 shows the high level steps involved. The invariants are symbolically executed in one stage while the second stage reads the explored valid results and simply proceeds with symbolic execution of the method under test. This enables a re-use of the symbolic exploration of the class invariants.

5.4 Abstract symbolic inputs

We define an *abstract symbolic input* as a tuple $\langle o, p \rangle$ where o is a rooted object graph and p is a path condition over fields in the object graph o. Figure 5.3 shows eight abstract symbolic inputs for a binary search tree with a bound of three nodes. Each abstract symbolic input consists of a concrete object graph with a path condition on its fields. We can generate these abstract symbolic inputs as:

```
SearchTree b;
if( !b.repOk())
  SEE_ignore();
```

Figure 5.3 *Abstract symbolic inputs* consist of an object graph/path condition pair.



SEE_ignore is a symbolic execution engine to backtrack the current search. When the above code is run using a symbolic execution engine it produces all valid object graphs along with their path conditions as shown in Figure 5.3.

An *abstract symbolic test driver* as a test function that utilizes abstract symbolic inputs and possibly other concrete and/or symbolic inputs to invoke a method under test. When the driver is symbolically executed, it uses abstract symbolic inputs to run *abstract symbolic tests*.

To test the add method of SearchTree, our running example, we can use the abstract symbolic test driver below.

```
void testSearchTreeAdd() {
  SearchTree* b=regenSymObjGraph<SearchTree>
                  (seedListOfSearchTree);
  try {
    // b already satisfies pre-conditions
    b->add (x); // x is a symbolic integer
    // check b using post-conditions
  } catch(...) {
    // report implementation bug
  }
}
```

5.5 Creating abstract symbolic inputs

To create abstract symbolic inputs, we need to save enough information to later reconstruct the symbolic execution state. A naive solution is to store the path condition as is. If the underlying symbolic execution engine supports pointers, then the path condition contains equality constraints for pointers involved in the object graph. For other symbolic execution engines, a scheme like lazy initialization [64] can be used. In this case, the path condition has equality constraints for integers that choose the pointer from a pool of pointers.

We can improve the naive solution by serializing the object graph as it exists during generation and remembering mapping of fields to variables in the path condition. This would be akin to storing it as shown in Figure 5.3. However, this makes regeneration complex. After instantiating the object graph, regeneration would require creating the symbolic variables involved in it and adding the constraints over them.

The above solution is practical, but there is an even easier approach that requires minimal changes in a symbolic execution engine to support abstract symbolic inputs. This approach forgoes storing the path condition and object graph separately (as shown in Figure 5.3) and only stores one solution to each path condition (e.g. $root = N_0, N_0.right = N_1, N_0.data = 1, N_1.data = 2$ etc.). Although, we lose information in this scheme, we show in the following section that when combined with the original class invariant, it is easy to reconstruct abstract symbolic inputs. It makes generating abstract symbolic inputs much easier.

To create abstract symbolic inputs for a class `C`, the user only provides a deterministic side-effect free class invariant (repOk). The following code is automatically generated and symbolically executed.

```
void generateC() {
  C c;
  if(!c.repOk())
    SEE_ignore();
}
```

SEE_ignore is a symbolic execution engine function that backtracks from the current search. Each completed path is solved for a satisfying solution (if one exists) and stored as an abstract symbolic input. Note that this is not the same as using concrete tests. This concrete representation is generated by a symbolic execution engine and when given to the same symbolic execution engine it can be used to retrace the execution path and regenerate the symbolic state. This is discussed in the next section.

As an additional benefit of storing a satisfying solution instead of a path condition, we have enabled the possibility to use other tools like Alloy [55] or Korat [10] for generating tests, while still using symbolic execution to realize them as abstract symbolic inputs. Similarly, a symbolic execution engine that supports pointers (e.g. CUTE [91]) or one with lazy initialization (e.g. JPF [112]) can be used for generation and another one can be used for execution that might better support unmodified large programs (e.g. KLEE [14]). Even the programming languages used for generation and execution predicates can be different, if the class invariant is written in both languages.

5.6 Regenerating symbolic execution state

To regenerate symbolic execution state from abstract symbolic inputs stored as a satisfying assignments to path conditions, we introduce a new algorithm. Symbolic execution engines did not need to be modified for generating abstract symbolic inputs because we stored them as a satisfying assignment, and symbolic execution engines depend on the ability to solve path conditions. However for reconstructing abstract symbolic inputs, we need to modify the symbolic execution engine. A naive solution to regenerate is to take the stored values as concrete and run the method under test using them. However, this defeats the purpose as we cannot explore all possibilities for the method under test that were possible using the path conditions. Thus we need to rebuild the symbolic state.

The technique we use utilizes *seeds*. Seeds are given to a symbolic execution engine to start its search. A seed consists of a tuple of values which provide the initial value for new symbolic variables formed. Some symbolic execution engines [42, 91] collect the path constraint for a complete execution using the seed values and then explore other branches by negating clauses in the path constraint. On the other hand, forward symbolic execution [14, 22, 65] checks on every branch that a satisfying seed exists to either prioritize that branch or exclude other branches altogether. We describe our technique for forward symbolic execution. It can be adapted for other tools as well.

Our algorithm in Algorithm 5.1 works in the context of a symbolic execution engine. When the code under test requests loading abstract symbolic inputs, the function REGENSYMOBJ (line 3) gets invoked. It takes the current symbolic state, a list of seeds where each seed is a solution to one path condition, and an object of the type to be generated. After this function returns, the fields of this object are constrained by satisfying the class invariant.

The *seed set* (set of solutions to path conditions – now to be used as seeds for new symbolic variables) in the current symbolic execution state should be empty (line 4). If it is not empty it means a previous abstract symbolic input was not

Algorithm 5.1: Regenerating abstract symbolic inputs — Invoked by the user to load abstract symbolic inputs.

```
globalPruningEnabled ← FALSE;
input  : state, seedList, symObject
assert(ISEMPTY(GETSEEDSET(state)));
forall the f ← seedList do
    addToSet(GETSEEDSET(state), f);
end
globalPruningEnabled ← TRUE;
executeSymbolic(symObject→repOk);
globalPruningEnabled ← FALSE;
```

Algorithm 5.2: Regenerating abstract symbolic inputs — Invoked by symbolic execution engine whenever a new branch is seen. Branch is pruned if it returns false. .

```
input  : state
if globalPruningEnabled then
    forall the f ← getSeedSet(state) do
        if isSolvableUsingSeed(state, f) then
            return TRUE;
        end
    end
    return FALSE;
end
return TRUE;
```

instantiated correctly.

On lines 5-7, we load the actual seeds from a list of seeds into the seed set of the current execution state. After that we enable pruning of unnecessary branches (line 8) and reset it to enable exploring everything (line 10) after symbolically executing the class invariant of the symbolic object being constructed (line 7).

The NEWBRANCH function (lines 13-23) is always called internally by the symbolic execution engine. If it returns false, the branch is pruned out. When pruning is enabled (line 14), it returns false (line 20) when there is no seed that satisfies the current execution state. New constraints are formed with each new branch, possibly resulting in some constraints becoming infeasible for every seed. Such branches are pruned out. These have been tested when the abstract symbolic input was generated and it turned out `repOk` returns false or they become infeasible. ISSOLVABLEUSINGSEED function (line 16) checks if the current path condition is necessarily false using the values in the given seed.

The changes required to support regenerating abstract symbolic inputs in a forward symbolic execution engine are: (1) a set of seed values which is used by new symbolic variables; (2) a function to decide if new branches are to be explored; and (3) the algorithm in Algorithm 5.1 to load seed values and temporarily enable branch pruning.

To reconstruct abstract symbolic inputs, the *user* invokes regenSymObjGraph in an abstract symbolic test driver (see Section 5.4). This invokes the internal REGENSYMOBJ function for the current symbolic execution state. KLEE [14] – the engine we used – supports seeds at the start of symbolic execution and allows restricting explored branches to these seeds. We modified it to enable adding seeds dynamically and restricting exhaustive branch exploration temporarily (for the duration of symbolically executing the class invariant).

5.7 Evaluation

We implemented staged symbolic execution on top of the KLEE symbolic execution tool [14]. To evaluate our approach, we consider five methods from a binary search tree, six methods from a sorted linked list, and two methods of a binary heap. We compare staged symbolic execution to standard symbolic execution using KLEE and black box bounded exhaustive testing using Korat. The experiments were performed on a 2.53GHz dual core i5 machine with 8GB of memory. We show how abstract symbolic tests enable small but effective test suites for methods with structurally complex arguments. We also show that testing time for a set of methods of the same type, and for methods with more than one structurally complex argument is substantially reduced. The following subsections go over different experiments we performed in detail.

5.7.1 One structurally complex argument

We test the `add` and `remove` methods of a binary search tree, a sorted linked list, and a binary heap – all taking a single structurally complex arguments (see first six method entries of Table 5.1).

Table 5.1 Comparison of standard symbolic execution and staged symbolic execution.

				Staged Symbolic Execution		
benchmark	max.	Total	Symbolic Execution	Stage 1	Stage 2	
	size[1]	non-equiv.	Valid/Explored	Valid/Explored	Valid/Explored	Stage 2
		tests[2]	tests (Time)	tests (Time)	tests (Time)	Savings
SearchTree.add	3	375	29/92 (34.2s)	9/72 (33.1s)	29/29 (9.0s)	3.8X
	4	5,955	99/344 (7m54.7s)	23/268 (6m26.5s)	99/99 (1m16.1s)	5.1X
	5	76,062	351/1298 (87m32.4s)	65/1012 (71m24.0s)	351/351 (12m29.2s)	5.7X
SearchTree.remove	3	375	49/112 (1m00.7s)	REUSED	49/49 (21.7s)	2.8X
	4	5,955	175/420 (13m48.2s)	REUSED	175/175 (3m42.1s)	3.7X
	5	76,062	637/1584 (151m52.2s)	REUSED	637/637 (38m37.8s)	3.9X
SortedList.add	6	12,012	28/56 (51.5s)	7/35 (44.9s)	28/28 (8.6s)	6.0X
	7	51,480	36/72 (3m02.4s)	8/44 (2m58.2s)	36/36 (15.0s)	12.2X
	8	128,790	45/90 (13m09.3s)	9/54 (14m01.6s)	45/45 (33.1s)	23.8X
SortedList.remove	6	12,012	28/56 (1m18.9s)	REUSED	28/28 (8.6s)	9.2X
	7	51,480	36/72 (4m23.4s)	REUSED	36/36 (16.4s)	16.1X
	8	128,790	45/90 (16m35.6s)	REUSED	45/45 (1m04.7s)	15.4X
BinaryHeap.add	8	6,937,713	26/135 (1m57.2s)	9/118 (1m41.7s)	26/26 (20.1s)	5.8X
	9	83,510,790	29/165 (3m06.6s)	10/146 (2m47.4s)	29/29 (27.6s)	6.8X
	10	988,213,787	32/198 (4m37.0s)	11/177 (4m28.2s)	32/32 (35.1s)	7.9X
BinaryHeap.remove	8	6,937,713	13/122 (2m18.2s)	REUSED	13/13 (23.1s)	6.0X
	9	83,510,790	14/150 (3m14.9s)	REUSED	14/14 (34.5s)	5.6X
	10	988,213,787	15/181 (4m49.4s)	REUSED	15/15 (41.5s)	7.0X
SearchTree.isPreOrder	2,2	375	16/70 (28.8s)	14/92 (44.0s)	16/16 (5.7s)	5.1X
	3,3	27,636	48/310 (8m25.4s)	13/86 (39.2s)	48/48 (58.7s)	8.6X
	4,4	2,623,995	149/1389 (157m10.9s)	28/288 (6m37.4s)	149/149 (11m23.8s)	13.8X
SearchTree.isEqual	2,2	625	16/96 (53.2s)	REUSED	16/16 (8.9s)	6.0X
	3,3	108,241	81/711 (46m37.5s)	REUSED	81/81 (4m47.1s)	9.7X
	4,4	28,100,601	TIMEOUT[3]	REUSED	529/529 (293m10.02s)	N/A
SortedList.merge	3,3	7,056	69/119 (1m05.3s)	REUSED	69/69 (15.8s)	4.1X
	4,4	245,025	251/341 (4m36.3s)	REUSED	251/251 (1m26.1s)	3.2X
	5,5	9,018,009	923/1070 (24m32.9s)	REUSED	923/923 (17m19.1s)	1.4X
SortedList.isEqual	4,4	245,025	55/145 (3m21.5s)	REUSED	55/55 (24.4)	8.2X
	5,5	9,018,009	91/238 (8m52.6s)	REUSED	91/91 (1m03.1s)	8.4X
	6,6	344,622,096	140/364 (25m08.0s)	REUSED	140/140 (2m02.9s)	12.3X
SortedList.isIntersection	2,2,2	21,952	112/190 (2m59.1s)	REUSED	112/112 (18.9s)	9.5X
	3,3,3	10,648,000	796/1006 (13m28.3s)	REUSED	796/796 (3m40.0s)	3.7X
	4,4,4	6,028,568,000	TIMEOUT[3]	REUSED	5201/5201 (60m09.6s)	N/A

[1] All sizes from 0 up to this size are generated.

[2] Total non-equivalent tests are a cross product of black box tests generated by Korat [10] for each argument using an integer data domain equaling the *total* number of integers involved.

[3] Timeout is set to 5 hours.

The "Stage 2 Savings" column shows the savings in time, in comparison to standard symbolic execution assuming that Stage 1 has been done for another method (e.g. we are testing *add* and we have Stage 1 results from testing *remove* or we are doing regression testing and we have Stage 1 results from a previous run). From the results we can see that staged symbolic execution saves us from exploring a number of states repeatedly and stage 1 results can be readily used for testing other methods.

5.7.2 Multiple independent arguments

The last five entries of Table 5.1 are methods with two and three structurally complex independent arguments. We consider `isEqual` and `isPreOrder` methods of binary search tree. The former considers two binary search trees, while the later takes a binary search tree and a sorted linked list. We also take `isEqual`, `merge`, and `isIntersection` methods from a sorted linked list. The first two take two sorted linked lists as arguments, while the last takes three sorted linked lists.

The added advantage of staged symbolic execution with multiple independent arguments is that Stage 1 can only be run using one argument, and Stage 2 can use the results repeatedly. Standard symbolic execution has to symbolically explore the class invariant of the second argument for every valid instance of the first argument.

Note that multiple dependent arguments are the same as single argument as they can be generated using a `repOk` that invokes the `repOk` of both arguments. We therefore do not evaluate them separately.

To show how staged symbolic execution scales with varying number of arguments, we consider a `union` method of a binary search tree and of a sorted linked list. We test taking the union of two, three, and four arguments and present the

Table 5.2 Increasing number of arguments.

# args	Total non-equiv. tests[1]	Symbolic Exec.		Staged Symbolic Exec.		
		Valid/Explored tests	(Time)	Valid/Explored tests	(Time)	Savings
SearchTree.union (size 2)						
2	625	16/96	(47.7s)	20/36	(11.6s)	4.1X
3	117,649	64/400	(14m18.9s)	68/84	(1m53.5s)	7.6X
4	43,046,721	256/1616	(244m23.5s)	260/276	(30m44.8s)	7.9X
SortedList.union (size 2)						
2	225	9/33	(9.7s)	12/18	(7.2s)	1.3X
3	21,952	27/105	(1m29.3s)	30/36	(19.0s)	4.7X
4	4,100,625	81/321	(10m50.9s)	84/90	(1m47.1s)	6.1X

[1] Total non-equivalent tests calculated as in Table 5.1.

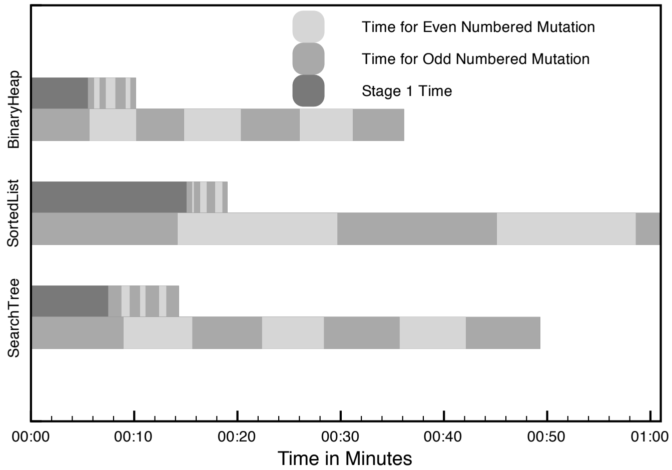
results in Table 5.2.

5.7.3 Regression testing using mutants

In this experiment, we show the benefit of staged symbolic execution for regression testing. Following previous work [31], we use mutants to simulate software evolution. We generated mutants manually with mutation operators: changing a comparison operator, changing a field with another field of the same type, and deleting a statement. (adapted from [83]). Six mutants of the `add` method are generated for each of binary search tree, sorted linked list, and binary heap.

For staged symbolic execution, we only need to run the first stage once because the pre-conditions of `add` method have not changed. Thus the cost of this stage is amortized over all the runs. We show our data in Table 5.3 and plot it as a graph in Figure 5.4. We call the original `add` method m_0 while the mutants are called $m_1 - m_6$. As we can see that the total time taken by staged symbolic execution is significantly less than a normal symbolic execution of all mutants. This shows the performance advantage of sharing symbolic execution results of one stage for regression testing.

Figure 5.4 Comparison of time for testing the original add method and six mutants for SearchTree (size 4), SortedList (size 8), and BinaryHeap (size 10).



5.7.4 Observations

Staged symbolic execution provides a number of benefits over standard symbolic execution and over test suites made using black box techniques. We observe:

Enabling symbolic test suites. The standard way to create test suites using standard symbolic execution is to concretize the tests. If this concretization is done using the class invariant alone, then we lose important tests that could have been created if the method under test was symbolically executed along with the class invariant. If however, the concretization is done after the method under test is symbolically executed, the test suite has to be regenerated whenever the method is changed. Staged symbolic execution requires a significantly smaller number of tests to be stored. It uses abstract symbolic inputs that can be saved and used to

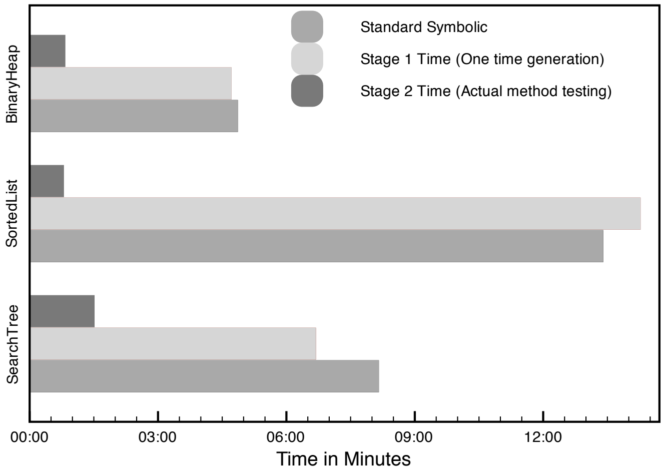
Table 5.3 Comparison of time for regression testing.

			Staged Symbolic Execution	
benchmark		Symbolic Exec.	Stage 1	Stage 2
		Valid/Explored Tests (Time)	Valid/Exp. (Time)	Valid/Explored Tests (Time)
Mutants of `SearchTree` `.add` (size 4)	m_0	99/344 (7m54.7s)	23/268 (6m26.5s)	99/99 (1m16.1s)
	m_1	45/290 (6m41.5s)		45/45 (49.7s)
	m_2	71/316 (6m43.5s)		71/71 (58.3s)
	m_3	64/309 (6m01.4s)		64/46 (33.2s)
	m_4	64/309 (7m18.3s)		64/64 (1m17.3s)
	m_5	64/309 (6m25.7s)		64/64 (44.0s)
	m_6	99/344 (7m12.7s)		99/99 (1m13.0s)
		Total: 48m17.8s	**Total:** 13m18.1s **Savings:** 3.6X	
Mutants of `SortedList.add` `.add` (size 8)	m_0	45/90 (13m09.3s)	9/54 (14m01.6s)	45/45 (33.1s)
	m_1	17/62 (15m30.0s)		17/17 (10.6s)
	m_2	24/69 (15m25.7s)		24/24 (34.6s)
	m_3	45/90 (13m28.2s)		45/45 (39.9s)
	m_4	45/90 (14m51.6s)		45/45 (47.1s)
	m_5	45/90 (15m27.4s)		45/45 (44.2s)
	m_6	45/90 (14m50.5s)		45/45 (39.0s)
		Total: 102m42.7s	**Total:** 18m10.1s **Savings:** 5.7X	
Mutants of `BinaryHeap` `.add` (size 10)	m_0	32/198 (4m37.0s)	11/177 (4m28.2s)	32/32 (35.1s)
	m_1	24/190 (4m32.9s)		24/24 (33.4s)
	m_2	28/194 (4m37.9s)		28/28 (35.3s)
	m_3	24/190 (5m30.6s)		24/24 (57.0s)
	m_4	24/190 (5m40.6s)		24/24 (57.7s)
	m_5	13/179 (5m09.2s)		13/13 (29.4s)
	m_6	25/191 (4m57.9s)		25/25 (32.3s)
		Total: 35m06.1s	**Total:** 9m08.4s **Savings:** 3.8X	

reconstruct the symbolic state at a later stage.

Performance improvements. We have seen the performance benefit of staged symbolic execution in Table 5.1. We plot some data from that table in Figure 5.5. We can see that Stage 1 and standard symbolic execution take more or less the same amount of time, whereas Stage 2 takes significantly less time than a complete run of symbolic execution.

Figure 5.5 Comparison of time for SearchTree (size 4), SortedList (size 8), and BinaryHeap (size 10).



Library of abstract symbolic tests. We have seen that we have isolated most of the overhead of symbolic execution for structurally complex inputs in the first stage of staged symbolic execution. This stage works regardless of the method under test. Also, the number of generated tests is manageable. This means, we can create a library of abstract symbolic tests. When an abstract symbolic test from such a library is used, there is no overhead at all and the first invocation gets all the benefit. Such a library can contain abstract symbolic tests for common structures like lists, trees, etc. The library can even be generated using other tools, e.g. black box tools like Alloy or Korat.

Chapter 6

Discussion

Our choice to develop the Pikse methodology in the context of Korat and symbolic execution was motivated by an initial empirical study that we performed. To enhance the applicability of the Pikse techniques for combined black-box and white-box testing, we developed a form of symbolic execution for Alloy. This chapter describes the empirical study (Section 6.1) and symbolic execution for Alloy (Section 6.2), which were initially presented at ICFEM 2009 [99] and ICFEM 2011 [97].

6.1 Empirical study

Structural constraint solving allows finding object graphs that satisfy given constraints, thereby enabling software reliability tasks, such as systematic testing and error recovery. Since enumerating all possible object graphs is prohibitively expensive, researchers have proposed a number of techniques for reducing the number of potential object graphs to consider as candidate solutions. These techniques analyze the structural constraints to prune from search object graphs that cannot satisfy the constraints. Although, analytical and empirical evaluations of individual techniques have been done, comparative studies of different kinds of techniques are rare in the literature. We performed an experiment [99] to evaluate the relative strengths and weaknesses of some key structural constraint solving techniques. The experiment considered four techniques using: a model checker, a SAT solver, two symbolic execution engines, and a specialized solver. It focused on their relative abilities in expressing the constraints and formatting the output object graphs, and most importantly on their performance. Our results highlight the trade-offs of different techniques and help choose a technique for practical use.

Generating test inputs for programs that manipulate structurally complex inputs like XML documents or red black trees is a complex operation. Manual generation of these tests is time consuming, error prone, and has fairly limited ability to find bugs whereas systematic testing, which is effective at finding bugs, is not straightforward as there are no simple enumerators for structurally complex inputs.

Automated generation of structurally complex test inputs can be done in two basic ways: using generator functions [113, 115] and by solving constraints [10, 76]. Generator functions are functions that perform basic operations to construct and build structures (e.g., constructors or mutator methods in Java). Automated testing using generator functions typically uses different orderings of generator functions to produce test inputs. This can however result in the same structures repeated, i.e., redundant tests, and some kinds of structures may never be produced. Generator functions are mostly applied for generating larger inputs effectively.

Automated testing by solving structural constraints [10, 76] enables *systematic testing* where the program is tested against all test inputs within given bounds. Even though doing so is feasible only for small bounds, it has been shown to give high code coverage and find faults in programs with structurally complex inputs [62, 76, 106]. Also, by writing constraints we can conveniently describe a whole class of structurally complex test inputs. In this dissertation, we discuss the techniques that can be used for systematic testing based on structural constraint solving.

The structural constraints used by systematic testing techniques are usually written either as declarative constraints or as imperative constraints. Alloy [55] (one of the techniques discussed here) uses declarative constraints written in relational logic using quantified formulas. The other three techniques that we evaluate use imperative constraints. We call them *imperative* in contrast to *declarative* as they use constraints written in an imperative language (C or Java in our case). We note that these imperative constraints are required to be free of side-effects and hence are declarative in nature (even though they are written in an imperative language).

For the purpose of comparison and explaining how constraints are written in different approaches, we will take red-black trees [7, 49] as our running example. We pick this representative example as it is one of the more complex structures, one of the structures commonly used for evaluation in previous work, and one that is likely to be familiar.

6.1.1 Background of Subject Tools

6.1.1.1 JPF — Model Checker

Model checking [20] has traditionally been applied to software [5, 25, 50, 111] for checking event sequences, specified in temporal logic or as a finite state machine of API usage rules. If a program is checked successfully, no input and execution can lead it to an error. Thus model checking provides a strong guarantee. However these techniques did not consider checking properties and validity of complex structures. The model checkers BLAST and SLAM are also used for white-box test input generation [8] targeting to cover specific predicates. The two are also not applied to solving complex structural constraints.

Generalized Symbolic Execution [64] introduced the idea of using a model checker for solving structural constraints. As an enabling technology, the JPF (Java Path Finder) model checker [111] was used. JPF is an explicit-state model checker for Java programs that has been used to find errors in a number of complex systems [4, 11, 85]. It is built on top of a custom Java Virtual Machine (JVM). Therefore it handles all standard Java features and in addition allows non-deterministic choices written as annotations. These annotations are added by method calls to class `Verify`. The following methods in this class are important:

- `randomBool()` returns a non-deterministic boolean value
- `random(n)` returns a non-deterministic integer in [0,`n`]
- `ignoreIf(cond)` makes JPF backtrack if `cond` is true

Generalized symbolic execution provides a source-to-source translation of a Java program such that it can be symbolically executed by any standard model checker that supports non-deterministic choice. The technique of generalized symbolic execution is based on *lazy initialization*, i.e. it initializes fields when they are first accessed during symbolic execution of a method. Due to this lazy initialization, the algorithm only executes program paths on non-isomorphic inputs. This can be used for systematic generation of structurally complex inputs by symbolically executing a predicate checking structural constraints.

Listing 6.1 shows parts of Red Black Tree predicate written for JPF. Note that all accesses to structure variables are through accessor functions. One accessor function for `header` is also shown. It non-deterministically picks one of the nodes that have already been used or one of the new nodes.

Recently, this technique has been optimized by making modifications to Java Path Finder [39]. However these optimizations are specific to one model checker, whereas the original technique can be used on any model checker.

Listing 6.1. Parts of Red Black Tree predicate written for JPF.

```
class RedBlackTree {
    ...
    static Node[] nodes;
    static int maxNode = 0;
    boolean header_accessed = false;
    Node header;
    Node header() {
        if (!header_accessed) {
            header_accessed = true;
            if (maxNode < nodes.length - 1) {
                maxNode++;
                int r = Verify.random(maxNode);
                if( r != maxNode )
                    maxNode--;
                header = nodes[r];
            } else header = nodes[ Verify.random( maxNode ) ];
        }
        return header;
    }
    boolean repOk() {
        if (header() == null)
            return false;
        Set<Node> visited = new java.util.HashSet<Node>();
        visited.add(header());
        LinkedList<Node> workList = new LinkedList<Node>();
        workList.add(header());
        while (!workList.isEmpty()) {
            Node current = workList.removeFirst();
            if (current.left() != null) {
                if (!visited.add(current.left()))
                    return false;
                workList.add(current.left());
            }
            if (current.right() != null) {
                if (!visited.add(current.right()))
                    return false;
                workList.add(current.right());
            }
        }
        if (visited.size() != size() || size() < LOWER_BOUND )
          return false;
        return repOkColors() && repOkKeys();
    }
}
```

6.1.1.2 Alloy — Using a SAT Solver

SAT solvers solve boolean formulas. To use SAT solvers for solving structural constraints, we thus need a language for writing structural constraints, a compiler to translate that language into a boolean formula, and a mapping from the solution of the boolean formula into a solution to the structural constraint.

Alloy [54] provides a declarative language for writing these constraints. It is based on parts of the Z specification [102]. The Alloy Analyzer [56] provides a fully automated tool to solve these constraints using a SAT solver. The latest version of Alloy Analyzer (4.1.10) works with many state-of-the-art solvers like BerkMin [44], MiniSat [101], SAT4J (Java implementation of MiniSat), and ZChaff [78]. Alloy analyzer provides a translation from the declarative language of Alloy with quantifiers to a boolean formula when given bounds. It then translates the solution back to the declarative language.

TestEra [63] builds on Alloy to translate the solutions further back into actual Java structures. TestEra also adds a layer on top of Alloy language to facilitate writing preconditions and postconditions, and allows test case generation based on preconditions and function validation using its postconditions as an oracle. However for the purpose of constraint solving alone, Alloy is sufficient. The Alloy to Java translator component of TestEra can be used to translate Alloy solutions into Java structures. The translation time is insignificant in comparison to the constraint solving time.

We show class invariant for red-black trees modeled in Alloy in Listing 6.2. Note that this completely models red black trees. Addition of a few more syntactic sugar like definition of `Node` etc is all that is needed to generate all possible red black trees within given bounds. This concise representation is one of the key benefits of using a declarative language. However the learning curve of declarative programming for programmers used to program in imperative languages often offsets this benefit. The bounds for Alloy are written as below:

Listing 6.2. Red Black Tree constraint written for Alloy.

```
 all e: rbt·root·*(left+ right) |
   // BT: distinct children
   ( no e·(left+ right) || e·left ≠ e·right ) &&
   // BT: acyclic
   ( e ! in e·^(left+ right) ) &&
   // BT: distinct parent
   lone e·~(left + right) &&
   // BST: ordered
   lt[ e·left·*(right+ left)·key, e·key ] &&
   gt[ e·right·*(right+ left)·key, e·key ] &&
   // RBT: red node has black children
   ( e·color in Red && some e·(left + right)
    ⇒ e·(left + right)·color in Black )

 all e, f: rbt·root·*(left+ right) |
   // RBT: all paths from root to NIL have same #
of black nodes
   (no e·left || no e·right) && (no f·left || no f·right) ⇒
    #{ p: rbt·root·*(left+ right) |
     e in p·*(left+ right) && p·color in Black } =
    #{ p: rbt·root·*(left+ right) |
     f in p·*(left+ right) && p·color in Black }
```

```
run test for 1 rbt, exactly 3 Node
```

The class invariant requires the tree to satisfy binary search tree properties and the additional properties of red-black trees mentioned in comments in Listing 6.2. The reader is referred to Jackson [54] for detailed discussion of Alloy operators and syntax and to Guibas [49] for red-black tree properties.

6.1.1.3 CUTE — Symbolic Execution

The idea of symbolic execution dates back at least three decades [65]. Traditional symbolic execution is a combination of static analysis and theorem proving. In symbolic execution, operations are performed on symbolic variables instead of actual data. On branches, symbolic execution is forked with opposite constraints on symbolic variables in each forked branch. At times, the constraints on symbolic variables can become unsatisfiable signaling unreachable code. Otherwise, end of the function is reached and a formula on symbolic variables is formed. A solution to this formula will give a set of values that will direct an actual execution along the same path.

Renewed interest in symbolic execution is seen in the last decade [13, 23, 36]. Generalized Symbolic Execution [64] extended the concept to concurrent programs and complex structures.

The main problem with symbolic execution is that for large or complex units, it is computationally infeasible to maintain and solve the constraints required for test generation. Larson and Austin [70] combined symbolic execution with concrete execution to overcome this limitation. Their approach was primitive as they used symbolic execution to make the path constraint of a concrete execution and find other input values that can lead to errors along the same path.

DART (Directed Automated Random Testing) [42] is one of the first tools to systematically combine symbolic execution and concrete execution. Similar to previous approach, they formed a path constraint during concrete execution. However after the execution, they backtrack on the path constraint by negating clauses, solve the new constraints, and re-run concrete execution expecting it to follow a new path. When it is not feasible to solve the modified constraints, they substitute random concrete values. Another simultaneous effort was EGT (Execution Guided

Test Cases) [16] using a similar approach. Lastly, CUTE (Concolic Unit Testing Engine for C) [91], another tool using similar approach, is the tool that we will be using here. It is the only tool that can handle pointers and complex structures.

The idea of using CUTE to generate test cases has been briefly discussed but not evaluated [91]. There, the authors considered `prev` pointers in a doubly linked list and discussed the order (big O) of candidates CUTE and Korat (discussed below) explore to find answers. In our evaluations we thoroughly cover this example among others. In particular, we discuss the constants involved (time of exploring one candidate) and constraint rewriting requirements to understand which approach is likely better in practical usage.

We show parts of the red-black tree constraint written in C for use in CUTE in Listing 6.3. The `NODES` variable is introduced to keep a count of nodes used. We break the loop when more than `UPPER_BOUND` nodes have been touched and `return false` if less than `LOWER_BOUND` nodes were touched during the execution. This is how we control the desired number of objects when generating structures in CUTE. Rest of the constraint is similar to what was shown in Listing 6.1.

6.1.1.4 Pex — Symbolic execution based on Z3

Pex [109] is another symbolic execution engine that is based on the Z3 constraint solver [79]. Pex builds upon other dynamic symbolic execution schemes like CUTE[91] and DART[42]. Pex can reason about *safe* .NET programs and a subset of the *unsafe* features of .NET. Here *unsafe* means that there are unverifiable memory accesses using pointer arithmetic.

A key benefit of Pex is its integration in the Visual Studio IDE and its testing frameworks. This makes Pex accessible to a large audience. Pex, by default, is focused on statement coverage and uses heuristics to increase it. For bounded ex-

Listing 6.3. Parts of Red Black Tree predicate written for CUTE.

```
int repOk( struct bintree* b ) {
  struct listnode* visited=0, *worklist=0;
  int NODES = 0;
  if( b->root == 0 )
      return 0;
  visited = newnode( b->root, visited );
  ++NODES;
  worklist = newnode( b->root, worklist );
  while( worklist ) {
    struct node* current = worklist->data;
    worklist = worklist->next;
    if( current->left ) {
      if( !addunique( visited, current->left ))
        return 0;
      ++NODES;
      worklist = newnode( current->left, worklist );
    }
    if( current->right ) {
      if( !addunique( visited, current->right ))
        return 0;
      ++NODES;
      worklist = newnode( current->right, worklist );
    }
    if( NODES > UPPER_BOUND )
      return 0;
  }
  if( b->size != vcount || NODES < LOWER_BOUND)
    return 0;
  return repOkColors(b) && repOkKeys(b);
}
```

haustive testing, we however required it to produce all tests for the same statements. We restricted Pex's exploration to given bounds using additional comparisons like we did for CUTE.

Parts of the red-black tree predicate written in C# for Pex are shown in Listing 6.4. To use this `repOk` for exploration we write the following method and use it to run Pex explorations. The parameter `FailuresAndUniquePaths` ensures that we get all unique paths and paths covering the same statements are not omitted.

```
[PexMethod(TestEmissionFilter =
  PexTestEmissionFilter.FailuresAndUniquePaths)]
public static void RedBlackTreeTest
  ([PexAssumeUnderTest] RedBlackTree target)
{
  PexAssume.IsTrue(target.repOk(3,3));
}
```

6.1.1.5 Korat — A Specialized Solver

Basics of Korat are described in detail in Section 2.2. We here show a portion of red-black tree constraint written for Korat in Java in Listing 6.5. We also show how bounds are given for Red Black Tree in Korat's finitization in Listing 6.6.

Korat has been optimized in a number of ways (Section 7.2). In this study, we use the original implementation of the Korat algorithm [10].

6.1.1.6 Research Questions

The effectiveness of bounded exhaustive testing (generating all test cases satisfying the constraints) has been previously shown in application to many real applications. Here we are concerned with different tools to generate these tests. Thus we are not concerned with the fault detecting capability of these tools, as this

Listing 6.4. Parts of Red Black Tree predicate written for Pex.

```
public bool repOk(int LOWER_BOUND, int UPPER_BOUND) {
  if (root == null)
    return false;
  Dictionary<Node, int> visited = new Dictionary<Node, int>();
  visited.Add(root, 0);
  Stack<Node> workList = new Stack<Node>();
  workList.Push(root);
  while (workList.Count != 0) {
    Node current = workList.Pop();
    if (current.left != null) {
      if (visited.ContainsKey(current.left)
          || visited.Count == UPPER_BOUND)
        return false;
      visited.Add(current.left, 0);
      workList.Push(current.left);
    }
    if (current.right != null) {
      if (visited.ContainsKey(current.right)
          || visited.Count == UPPER_BOUND)
        return false;
      visited.Add(current.right, 0);
      workList.Push(current.right);
    }
  }
  if (visited.Count != size || size < LOWER_BOUND)
    return false;
  return repOkColors() && repOkKeys();
}
```

Listing 6.5. Parts of Red Black Tree predicate written for Korat.

```java
    public boolean repOK() {
        if (root == null)
            return false;
        Set<Node> visited = new HashSet<Node>();
        visited.add(root);
        LinkedList<Node> workList = new LinkedList<Node>();
        workList.add(root);
        while (!workList.isEmpty()) {
            Node current = workList.removeFirst();
            if (current.left != null) {
                if (!visited.add(current.left))
                    return false;
                workList.add(current.left);
            }
            if (current.right != null) {
                if (!visited.add(current.right))
                    return false;
                workList.add(current.right);
            }
        }
        if (visited.size() != size)
            return false;
        return repOkColors() && repOkKeys();
    }
```

Listing 6.6. Korat’s specification of bounds for Red Black Tree.

```
IFinitization f = FinitizationFactory.create(RedBlackTree.class);

IClassDomain entryDomain = f.createClassDomain(Node.class, numEntries);
IObjSet entries = f.createObjSet(Node.class, true);
entries.addClassDomain(entryDomain);

IIntSet sizes = f.createIntSet(minSize, maxSize);
IIntSet keys = f.createIntSet(−1, numKeys − 1);
IIntSet colors = f.createIntSet(0, 1);

f.set(”root”, entries);
f.set(”size”, sizes);
f.set(”Node.left”, entries);
f.set(”Node.right”, entries);
f.set(”Node.color”, colors);
f.set(”Node.key”, keys);
```

capability would be equal (given sufficient time) for all tools in our scenario. We are rather concerned with how to write the tests and interpret the output and most importantly how much time it takes to generate the tests.

We pose the following research questions for our experiment and analysis:

- What are the pros and cons of different tools in writing constraints and defining bounds?
- How is the output of a tool represented and how it can be converted into actual test inputs?
- What are the fastest tools for practical sizes of subject structures?
- How well do the tools perform with more and more complex constraints?
- What are the best tools in terms of time complexity?

Next we describe our experiment and its analysis.

6.1.2 The Experiment

6.1.2.1 Experimental Subjects

To evaluate the selected tools, we consider six data structures: three list structures, and three tree structures. Note that these complex structures are the foundation of several data structures used in applications. For example, an XML document, a file system hierarchy, Java or C class hierarchies, expression trees, abstract syntax trees for compiler can all be viewed as trees and are likely to give similar performance to one of the tree structures we consider here. We evaluate the following six structures:

1. Binary Tree
2. Binary Search Tree
3. Red Black Tree
4. Singly Linked List
5. Doubly Linked List
6. Sorted Linked List

Note that a red-black-tree is a binary search tree which is in turn a binary tree. By considering all three of them, we intend to learn the effect of increasing constraint complexity on tool performance.

To reduce bias, we took constraints for the above subjects from previous work [10], where available. In some cases, we needed to change the constraints so that the tool under evaluation performs bounded exhaustive testing (as discussed in the previous section).

6.1.2.2 Experimental Design

The experiment focused on:

1. Structurally complex constraints (6 constraints of subjects given in previous section)
2. Bounds (we considered 4 bounds for each subject structure)
3. The constraint solver (one of the four constraint solvers discussed in this dissertation)

On each run, we measured:

1. Time taken to generate all tests
2. Candidates generated to see isomorphism pruning

We also measure qualitative results for:

1. How constraints needed to be converted to run the tool
2. How bounds needed to be converted to run the tool
3. How results from the tool needed to be converted to test cases

Results reported for the experiment were averages of 10 repeated measurements. Thus, for each subject structure and each constraint solver and each given bounds, we ran the tool 10 times and computed the average. The experiments were performed on a Linux machine with Intel Pentium 4 2.8Ghz processor and 4GB RAM. The experiments for Pex were run on a Windows based machine with the same hardware.

6.1.2.3 Threats to Internal Validity

Threats to internal validity are influences that can affect dependent variables without researcher's knowledge. In this respect, our concerns include the way constraints are written and language differences. Constraints can be written to suit one tool and not the other. We have done our best effort is writing the constraints so that every tool can perform at its best. Language differences matter because one of the tools works in C while the rest work in Java. C implementations are inherently faster so the results of this tool would have a slight edge because of language. However this concern would have been more significant if this tool turned out to be the fastest which is not the case as we see below. Lastly, we had to use two different machines as Pex needed a windows machine. This can theoretically affect the experiment however we expect the OS performance to not be a major factor.

6.1.2.4 Threats to External Validity

Threats to external validity are conditions that limit us in generalizing the results of our experiment. Our biggest concerns in this area is that the subject programs might not be representative of complex constraints. To control this threat, we have studied literature regarding the tools and summarized the complex constraints previously studied, we have also studied structures discussed in algorithm books, and have found that the most commonly used complex structures are actually the basis of a large class of data structures. For example, B-trees, AVL trees, Sparse matrices, hash tables are all basically trees or a combination of trees and lists. We considered complex inputs of real programs like compilers (abstract syntax tree), XML parsers (XML Tree), web browser (HTML Tree), File system tree, Java class hierarchies, and expression trees. All of these share constraints with the basic structures we test here. Therefore we believe that our subjects are representative of complex constraints and can be used to evaluate constraint solvers.

6.1.2.5 Threats to Construct Validity

Threats to construct validity are situations where measurement instruments do not adequately capture concepts that they are supposed to capture. In this experiment, we measure performance and ease of writing constraints and using results. Measuring performance is always risky on today's multitasking machines. We controlled this threat with repeated measurements and with no sharing of resources. The quantitative analysis about constraint writing is more prone to this threat. We control this threat by providing raw data (how constraints are written, bounds given, results converted) and add our analysis on top of it.

6.1.2.6 Analysis Strategy

We summarize all the data first. We then make observations on this data and our observations on the three quantitative criteria of constraint writing, giving bounds, and using results. Finally, we show several comparisons between performance of different techniques in graphical form.

6.1.3 Data and Analysis

We provide performance comparison and its analysis followed by quantitative analysis.

6.1.3.1 Performance Comparison

Table 6.1 shows the results of our experiments. The first column lists the complex structures we chose. The next column specifies the size we are using. For

Binary Tree, Singly Linked List, and Doubly Linked List, we generate structures up to given size while we generate structures of exactly that size for the other three structures. The reason for this is that when generating structures with valid integer ranges of some data variables (e.g. Sorted List), then all tools except CUTE will produce all valid assignments while CUTE will provide a single valid assignment. This makes comparison difficult. We thus chose a fixed size and fixed range of integers such that only one valid assignment exists. The next four columns in the table list the times taken by each tool.

Alloy ran into solver limitations for sizes greater than about 15 nodes for all list structures. Similarly CUTE faced symbolic execution limitations for red black trees. Other numbers not available are time outs for the allocated 15 minutes.

Table 6.2 shows how well the candidate tools performed in terms of pruning isomorphic candidates. Korat and JPF never produced an isomorphic result. Also from their algorithm, they would never produce a normal isomorphic result according to the definition given previously. Note that their can be domain specific isomorphic results (e.g. isomorphic graphs) which no tool identifies as isomorphic. CUTE produced isomorphic candidates only when it ran into symbolic execution limitations. This happened in our case for red-black trees. Alloy produced isomorphic candidates most often. Its isomorphism pruning is most limited. For example, for a singly linked list, other than the root node and the tail node, it produces more than one isomorphic orderings of the middle nodes.

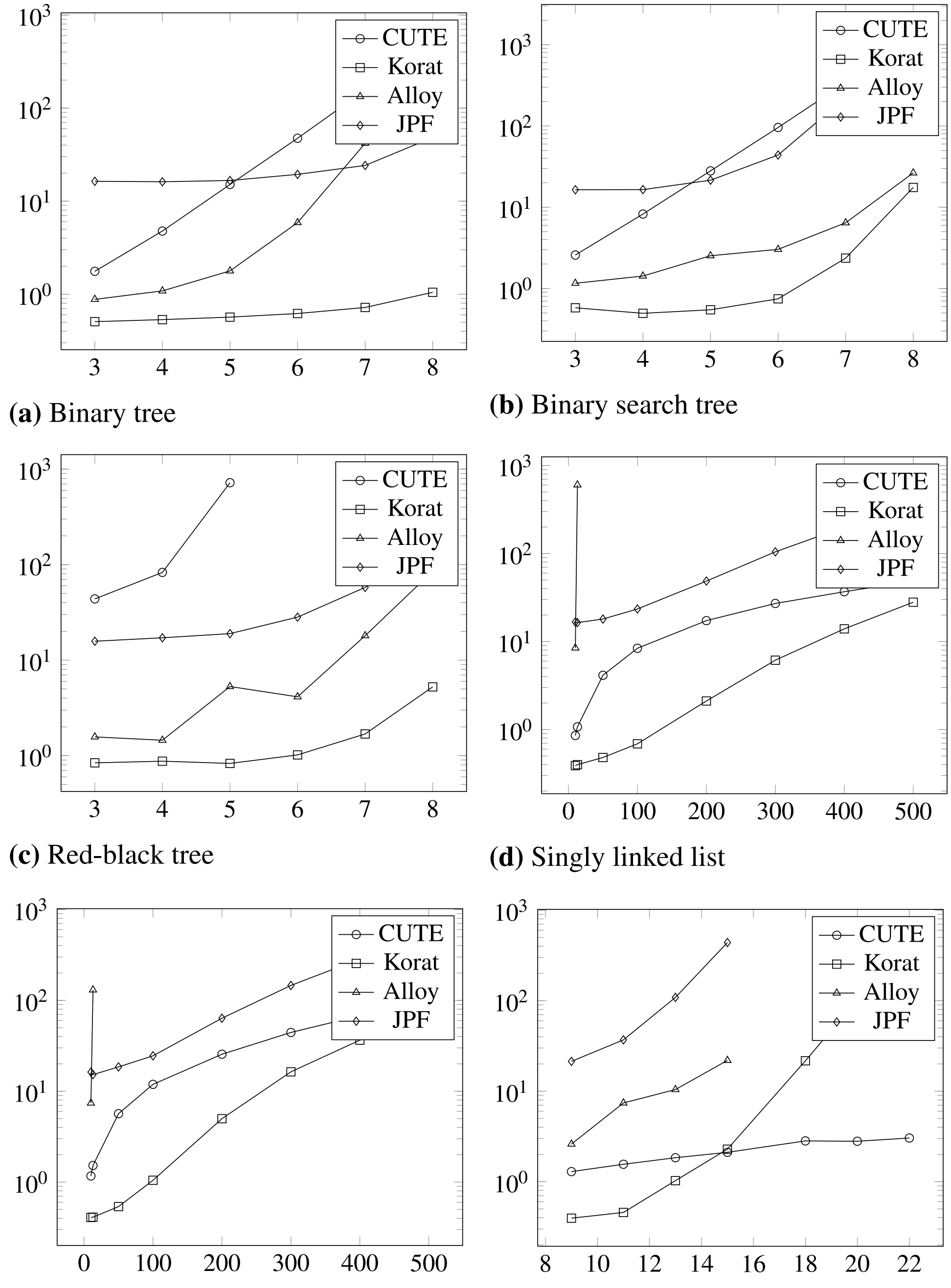
Lastly, Figure 6.1 shows six graphs, one for each subject structure and plots the performance of all four tools. The time axis is logarithmic since bounded exhaustive testing is an exponentially growing problem and a logarithmic scale better shows how the tools are performing.

We observe that other than sorted lists, Korat is the fastest tool within 1000s

Table 6.1 Results of generating bounded exhaustive test cases for six subject structures by CUTE, Pex, Korat, Alloy, and JPF. Time out or tool limitations are represented by a hyphen (-).

Subject	Size	CUTE	Korat	Alloy	JPF	Pex
Binary Tree	3	1.761	0.507	0.880	16.349	1.069
	4	4.774	0.533	1.085	16.158	1.483
	5	15.104	0.567	1.779	16.678	5.279
	6	47.427	0.620	5.882	19.405	18.725
	7	156.368	0.720	41.866	24.197	66.760
	8	527.292	1.048	520.868	48.389	259.729
Search Tree	3	2.580	0.579	1.159	16.415	0.890
	4	8.240	0.495	1.423	16.478	3.605
	5	28.015	0.547	2.529	21.498	13.803
	6	95.764	0.746	3.032	43.905	54.418
	7	341.444	2.363	6.437	222.893	217.971
	8	-	17.515	26.456	-	-
Red Black Tree	3	43.769	0.841	1.571	15.775	1.903
	4	82.905	0.875	1.450	17.139	5.718
	5	-	0.829	5.293	18.948	23.735
	6	-	1.018	4.132	28.186	97.294
	7	-	1.687	18.036	57.800	406.256
	8	-	5.250	85.277	170.962	-
Singly Linked List	10	0.855	0.389	8.452	16.661	0.400
	13	1.073	0.399	602.250	16.414	0.530
	50	4.136	0.481	-	18.015	6.038
	100	8.383	0.688	-	23.433	17.318
	200	17.273	2.110	-	48.625	71.786
	300	27.082	6.138	-	104.517	187.063
	400	36.811	13.939	-	200.062	378.176
	500	48.849	27.982	-	344.724	-
Doubly Linked List	10	1.167	0.408	7.408	16.221	1.475
	13	1.523	0.411	130.423	15.242	2.498
	50	5.657	0.537	-	18.511	83.219
	100	11.900	1.047	-	24.547	601.229
	200	25.538	4.987	-	63.614	-
	300	44.332	16.354	-	146.015	-
	400	67.828	36.503	-	285.589	-
	500	100.057	72.686	-	501.617	-
Sorted List	9	1.292	0.395	2.602	21.333	4.173
	11	1.557	0.457	7.409	36.900	7.000
	13	1.839	1.026	10.420	108.670	10.593
	15	2.110	2.286	21.874	439.063	15.298
	18	2.821	21.646	-	-	25.155
	20	2.797	102.609	-	-	33.483
	22	3.036	499.276	-	-	43.443

Figure 6.1 Performance Comparison of techniques for all six subject structures. Y-axis shows time in seconds on a logarithmic scale. X-axis shows size of structure.



(a) Binary tree

(b) Binary search tree

(c) Red-black tree

(d) Singly linked list

(e) Doubly linked list

(f) Sorted linked list

Table 6.2 Isomorphic candidates produced.

Subject	CUTE	Korat	Alloy	JPF
Binary Tree	NO	NO	**YES**	NO
Binary Search Tree	NO	NO	NO	NO
Red Black Tree	**YES**	NO	NO	NO
Singly Linked List	NO	NO	**YES**	NO
Doubly Linked List	NO	NO	**YES**	NO
Sorted List	NO	NO	NO	NO

time. For binary tree and Red Black Trees, it also seems to grow the slowest. For Binary Trees and Binary Search Trees, CUTE is growing linear on a logarithmic scale which means it is slightly better in terms of time complexity but the actual problem size where it would take over Korat would be huge.

CUTE is the only tool that handles Sorted Lists successfully, It touches our 1000s limit for generating about 500 element lists. This huge difference is because the other tools internally generate all possible combinations (n!) whereas symbolic execution does not. This is also the motivation around some recent work on Korat and JPF to use symbolic execution for primitives and use the native algorithm for non-primitive fields [113].

Note also in all graphs that CUTE has the best time complexity. It grows exponentially (trees) and sub-exponentially (lists) except for red black trees where symbolic execution faced limitations. Thus when symbolic execution faces limitations and CUTE reverts to take help from concrete execution, we may not get results comparable to other tools. This is one of the key weak points of CUTE for bounded exhaustive generation.

Alloy shows an interesting behavior. It performs better for Binary Search Trees (more complex constraint) than Binary Trees. We believe that this is because SAT solvers solve the easiest clauses first and the former gives it a better chance at doing that. Red black tree performance is in the middle and is better for 4 nodes than for 3 (and 6 nodes than for 5). We again believe this has to do with the formation of clauses.

If we carefully note, the graph of JPF is almost at a constant distance above Korat. Indeed, JPF structural constraint solving algorithm and the Korat algorithm principally make the same decisions. JPF is only burdened with running a model checking virtual machine and keeping a lot of additional state which Korat can do without. That is why they have similar time complexity but a different multiplier. Thus we can say that Korat is a much faster specialized implementation of what the JPF structural constraint solving algorithm does without the added overheads of model checking.

6.1.3.2 Qualitative Comparison

One of the research goals of our experiment was to discuss some qualitative differences between subject tools. We give summarized results in Table 6.3 and give a more detailed discussion of each difference below.

Constraint Writing: All tools except Alloy required constraints written in an imperative language. Constraints are required to be free of side-effects. CUTE constraints needed some tweaking to allow symbolic execution to explore all paths. For example, a `return size == 0` statement has to be changed to a branch statement with separate returns. JPF and Korat can use an arbitrary imperative function that is free of side-effects. Alloy required declarative predicates. Declarative specifications are concise and can be significantly smaller than an equivalent imperative specification. The trade-off is the learning curve of declarative language for programmers used to writing code in imperative languages.

Table 6.3 Comparison of structural constraint solving techniques on non-performance metrics.

	Constraints	Bounds	Output
CUTE	Imperative function: Some special care at branches to enable symbolic execution to visit both branches	For linear structures, giving a depth bound in invoking CUTE is enough; for others, special checks needed to be inserted inside the predicate	Each complex structure is available at end of testing function in a separate process
Korat	Imperative function: No special restrictions	An imperative function listing bounds for each object and predicate involved (*finitization*)	Each structure is available in a special function in the same single process
Alloy	Declarative predicate: In relational quantified logic	List of bounds for each object involved	Result is a list of solutions that can be translated into actual heap structures using Alloy to Java translator in TestEra [63]
JPF	Imperative function: Need to use special accessor functions (can be added automatically) that use model checker's non-determinism	Ranges can be specified in special accessor functions	Each complex structure is available at end of testing function in a separate process

Giving Bounds: Korat and Alloy were the easiest to provide bounds, which is not surprising since they are designed for specification-based, bounded exhaustive checking. They differed in that Alloy required bounds for each *type* whereas Korat was more explicit in requiring bounds for each *field* of each type. Also for primitives, Korat can use lower bounds and upper bounds whereas Alloy would need those bounds as part of specification and not as part of bounds. To limit structures generated by CUTE within bounds, we needed to tweak its imperative predicate. Providing bounds using the JPF approach was simple. In this approach the required arrays (universe of values) were constructed during the testing `Main` method. Values of these arrays are non-deterministically used by accessor functions (possibly automatically added).

Using Results: The JPF approach and CUTE approach produce each result, i.e. structure that represents a test input, in a separate execution (process). This result can directly be used for testing or saved for later use. Korat approach produces each result in the same execution (process). The result can be saved. Direct testing has to be careful about using a new process to avoid crashing of Korat due to faulty code. In previous work, these results have been distributed for parallel test execution [77]. Alloy produces solutions to declarative specifications. These need to be converted to the corresponding imperative language for actual test use. One tool in this area is Alloy to Java converter used in TestEra [63]. This tool can generate actual Java structures corresponding to Alloy output.

Treatment of primitive fields: While the key benefit of structural constraint solving is non-primitive fields (pointers to objects), primitive fields also pose a limitation. All the surveyed tools except CUTE try all possible values for a given primitive field. This often results in exponential or factorial amount of time. CUTE excels in this area by providing a single valid solution for such fields.

6.1.4 Summary and Conclusions

In this dissertation, we performed an empirical study of using four different techniques for constraint solving to perform bounded exhaustive testing. Bounded exhaustive testing has been previously shown effective at finding faults in real programs. Here, our goal is to compare the performance of these tools. We considered the CUTE tool based on symbolic execution, the JPF model checker, the Alloy tool based on SAT, and the specialized solver Korat . Our key results are:

- The fastest tool for most of the subjects of small size is Korat. However it degrades in performance when several constraints are on primitive fields.
- The JPF constraint solving approach using lazy initialization is effectively a slower Korat.

- Alloy provides the most concise way of writing predicates. For programmers knowledgeable in declarative languages, it can significantly reduce time to write or maintain specifications.

- CUTE provides better time complexity than most tools however the slope constant is fairly high. This is because of the symbolic execution overhead.

- CUTE requires some tweaking of class invariants to enable bounded exhaustive generation.

- No tool gives better non-isomorphic generation for exhaustive enumeration than the Korat algorithm (and likewise lazy initialization using JPF).

- All tools except CUTE provide bounded exhaustive checking by design and CUTE focuses on generating one input per path.

6.2 Symbolic Alloy

While symbolic execution today lies at the heart of some highly effective and efficient approaches for checking imperative programs, the use of symbolic execution in declarative programs is uncommon. Unlike imperative programs that describe *how* to perform computation to conform to desired behavioral properties, declarative programs describe *what* the desired properties are, without enforcing a specific method for computation. Thus, symbolic execution, or execution per se, does not directly apply to declarative programs the way it applies to imperative programs.

In this section, we present symbolic Alloy [97] – a novel approach to symbolic execution for declarative programs written in the Alloy modeling language [55]. Unlike imperative programs that describe *how* to perform computation to conform to desired behavioral properties, declarative programs describe *what* the desired properties are, without enforcing a specific method for computation. Thus, symbolic execution does not directly apply to declarative programs the way it applies to imperative programs. Our insight is that we can leverage the fully automatic, SAT-based analysis of the Alloy Analyzer to enable symbolic execution of Alloy models – the analyzer generates instances, i.e., valuations for the relations in the model, that satisfy the given properties and thus provides an execution engine for declarative programs. We define symbolic types and operations, which allow the existing Alloy tool-set to perform symbolic execution for the supported types and operations. We demonstrate the efficacy of our approach using a suite of models that represent structurally complex properties. Our approach opens promising avenues for new forms of more efficient and effective analyses of Alloy models.

Our insight into symbolic execution for Alloy is that path conditions in symbolic execution, which by definition are constraints (on inputs), can play a fundamental role in effective and efficient analysis of declarative programs, which themselves are constraints (that describe "what"). The automatic analysis performed by the Alloy tool-set enables our insight to form the basis of our approach. Given an Alloy model, the analyzer generates *instances*, i.e., concrete valuations for the sets and relations in the model, which satisfy the given properties. Thus, the analyzer, in principle, already provides an *execution engine* for declarative programs, which bears resemblance to concrete execution of imperative programs. Indeed, a common use of the analyzer is to *simulate* Alloy predicates and iterate over concrete instances that satisfy the predicate constraints [55]. The novelty of our work is to introduce *symbolic execution* of Alloy models, which is inspired by traditional symbolic execution for imperative programs. Specifically, we introduce symbolic types and symbolic operators for Alloy, so that the existing Alloy Analyzer is able to perform symbolic execution for the supported types and operations. To illustrate, symbolically simulating an Alloy predicate using our approach allows generating a *symbolic instance* that consists of a concrete valuation, similar to a traditional Alloy instance, as well as a symbolic valuation that includes a constraint on symbolic values, similar to a path condition.

We demonstrate the efficacy of our approach using a suite of models that represent a diverse set of constraints, including structurally complex properties. Our approach opens promising avenues for new forms of more efficient and effective analyses of Alloy models. For example, our approach allows SAT to be used to its optimal capability for structural constraint solving, while allowing solving of other kinds of constraints to be delegated to other solvers. As another example, our approach allows Alloy users to view multiple instances simultaneously without the need for enumeration through repeated calls to the underlying solver: a symbolic instance represents a class of concrete instances.

6.2.1 Illustrative Example

This section presents an example of symbolic execution of Alloy formulas using a sorted linked list. In Section 2.3 we presented the Alloy specification for a linked list. To make it into a sorted linked list, we use the following predicate.

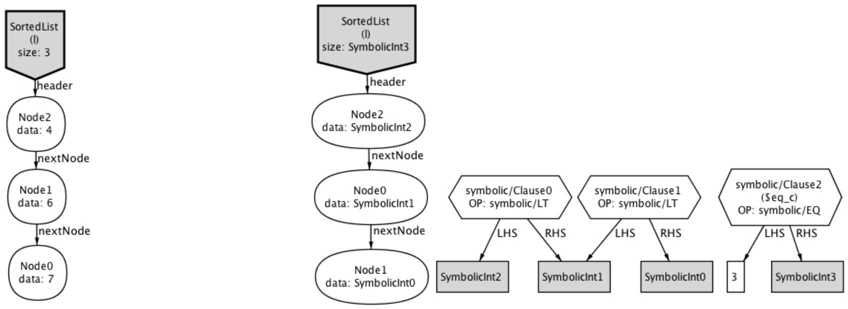
```
pred RepOk(l: SortedList) {
  all n: l·header·*nextNode| n !in n·^nextNode -- acyclicity
  #l·header·*nextNode = l·size -- size ok
  all n: l·header·*nextNode |
    some n·nextNode ⇒ n·data < n·nextNode·data } -- sorted
```

The Alloy Analyzer can be used on this model to find instances of a sorted linked list. We add the following commands to our model to test the `RepOk` predicate for three nodes.

```
fact { SortedList·header·*nextNode = Node } -- no unreachable Node
run RepOk for 3 -- maximum 3 atoms of each kind
```

As a result of executing this model, Alloy Analyzer produces an instance. The user can get more and more instances by clicking next. One example instance is shown in Figure 6.2a. This sorted linked list represents the sequence $\langle 4, 6, 7 \rangle$. The Alloy Analyzer produces many more instances with three nodes with different sorted arrangements of integers in the domain of Alloy integers.

Figure 6.2 Visualizing a sorted linked list with three nodes.



(a) Concrete **(b)** Symbolic

Symbolic execution of Alloy (Section 6.2.2) is a technique to produce instances with symbolic variables and a set of constraints over those symbolic variables. These individual constraints are called *clauses* in our models. The technique is implemented as (1) Alloy library module, (2) a set of guidelines for the user on how to write their Alloy formulas, (3) a set of mechanically generated rules, and finally (4) a mechanism to invoke the Alloy Analyzer. To use symbolic execution of Alloy, the user writing the model has to include the `symbolic` module:

```
open symbolic
```

The user changes any uses of `Int` they wants to make symbolic to `SymbolicInt`. The updated signature declarations for our example look like this:

```
one sig SortedList {
  header: lone Node,
  size: SymbolicInt }
sig Node {
  data: SymbolicInt,
  nextNode: lone Node }
```

Lastly, the user changes any operations performed on the symbolic variables to use the predicates provided by the `symbolic` module (e.g. `eq`, `lt`, `gt`, etc.). We follow the predicate names introduced by the Alloy 4.2 release candidate in its `integer` module for concrete operations on integers. If the user uses these predicates, no changes are required for predicate invocation. The Alloy Analyzer's type checking finds out whether a symbolic operation is needed or a concrete operation. The updated `RepOk` predicate looks like this:

```
pred RepOk(l: SortedList) {
  all n: l·header·*nextNode| n !in n·^nextNode -- acyclicity
  (#l·header·*nextNode)·eq[l·size] -- size ok
  all n: l·header·*nextNode |
    some n·nextNode ⇒ (n·data)·lt[n·nextNode·data] } -- sorted
```

As the next step, the Alloy module is transformed and a new `fact` is mechanically generated. This fact ensures that the symbolic integers used are all unique. For sorted linked list, this fact is:

```
fact {
  #SymbolicInt = (#SortedList)·plus[#Node]
  SymbolicInt = SortedList·size + Node·data }
```

Finally, when this updated model is run through Alloy Analyzer, models with a set of constraints on these symbolic integers are generated. An example instance with three nodes is given in Figure 6.2(b). This time, however, it is the only instance with three nodes. Other instances either have fewer or more nodes.

This helps the user visualize the model in a more efficient manner. Also a symbolic instance more explicitly states the relationship between data nodes.

6.2.2 Symbolic execution of Alloy formulas

This section presents the four key parts of our approach: (1) Alloy library module that introduces symbolic variables and operations on them as well as a representation for clauses that define constraints on symbolic fields, (2) changes required in the user model to introduce symbolic fields, (3) mechanically generated facts that enable consistent usage of symbolic values, and (4) Alloy Analyzer usage to restrict any redundant clauses from being generated.

6.2.2.1 Symbolic Alloy module

This section presents the Alloy module that enables symbolic execution. The module starts by the module declaration and a few signatures:

```
module symbolic

abstract sig Expr {}
sig SymbolicInt, SymbolicBool extends Expr {}

abstract sig RelOp {}
one sig lt, gt, lte, gte, eq, neq, plus, minus extends RelOp {}
```

`Expr` atoms represent expressions that can be symbolic variables or expressions on symbolic variables and plain integers. `RelOp` are single atoms (because of the `one` modifier) that represents a few binary operations we demonstrate. Next we define the `Clause` atom, which is an expression combining two symbolic variables, standard Alloy integers, or expressions.

```
abstract sig Clause extends Expr {
  LHS: Expr+ Int,
  OP: RelOp,
  RHS: Expr+ Int }
```

Next, we have a set of predicates that require certain clauses to exist. For example the following `lt` and `eq` predicates would require that appropriate `Clause` atoms must exist. These `Clause` atoms in the final output show us the relationship enforced on symbolic variables in the model.

```
pred lt(e1: Expr+ Int, e2: Expr+ Int) {
  some c: Clause | c·LHS = e1 && c·OP = LT && c·RHS = e2 }
pred eq(e1: Expr+ Int, e2: Expr+ Int) {
  some c: Clause | c·LHS = e1 && c·OP = EQ && c·RHS = e2 }
```

Similar predicates exist for all supported operations and Alloy functions exist to combine `plus` and `minus` operators to form more complex expressions.

6.2.2.2 User modifications to Alloy model

This section describes the changes required of the user in their model. Some such changes were discussed in Section 6.2.1 in the context of a sorted linked list.

The first change is a call to use the `symbolic` module. This imports the library signatures, predicates, and functions discussed in the previous section.

```
open symbolic
```

Next the user changes `Int` to `SymbolicInt` and `Bool` to `SymbolicBool`. These are the only primitive types supported by the Alloy Analyzer and we enable symbolic analysis for both of them.

Lastly, the user has to change all operations on symbolic variables to use one of the predicates or functions in the `symbolic` module. However, the names we used are the same as those used in the built-in Alloy `integer` module. The new recommended syntax of Alloy 4.2 release candidate is already to use such predicates. Specifically, for `plus` and `minus` predicates, the old syntax is no longer allowed. The `+` and `-` operators exclusively mean set union and set difference now.

We follow the lead of this predicate-based approach advocated in the Alloy 4.2 release candidate and support `eq`, `neq`, `lt`, `gt`, `lte`, `gte`, `plus`, and `minus` in our `symbolic` module. If the user is using old Alloy syntax, he has to change to the new syntax as follows:

```
a = b  ⇒  a.eq[b]
a < b  ⇒  a.lt[b]
a > b  ⇒  a.gt[b]
a + b  ⇒  a.plus[b]
a - b  ⇒  a.minus[b]
```

The `plus` and `minus` operations in our `symbolic` library come in two forms: as a predicate and as a function. The predicate requires the clause to exist and the function returns the existing clause. For example, to convert an expression `a+b>c` the user first converts it to new syntax i.e. `(a.plus[b]).gt[c]`. Then he adds the plus operation as a separate predicate as well i.e. `a.plus[b] && (a.plus[b]).gt[c]`. The compiler recognizes the first invocation as a predicate that requires a new clause to exist and the second invocation as returning that clause. If the predicate is omitted, the function returns no clause and no satisfying model is found. We include two case studies that show how it is used (Figure 6.5 and Figure 6.6).

6.2.2.3 Mechanically generated facts

This section presents the Alloy facts that our technique mechanically generates to ensure soundness of symbolic execution. These facts ensure that symbolic variables are not shared among different objects. For example, two `Node` atoms cannot point to the same `SymbolicInt` atom as `data`. Otherwise, we cannot distinguish which node's symbolic variable a Clause is referring to. Note that this does not prevent two nodes to contain the same integer value.

We use two mechanically generated facts to ensure uniqueness of symbolic variables. To form these facts, we find all uses of symbolic variables (SymbolicInt and SymbolicBool). We describe the generation of facts for SymbolicInt. Similar facts are generated for SymbolicBool.

Consider a sig A where B is a field of type SymbolicInt – i.e. B is a relation of the type A→SymbolicInt. We form a list of all such relations {(A1, B1), (A2, B2), (A3, B3), ...} and then generate two facts.

The first fact ensures that all SymbolicInt atoms are used in one of these relations and the second fact ensures that we exactly have as many SymbolicInt atoms as needed in these relations. If any SymbolicInt atom is used in two relations, then some SymbolicInt atom is not used in any relation (because of second fact), but unused SymbolicInt atoms are not allowed (because of first fact). Thus the two facts are enough to ensure unique symbolic variables.

```
SymbolicInt = A1·B1 + A2·B2 + A3·B3 + ···
#SymbolicInt = #A1 + #A2 + #A3 + ···
```

Note that if some sig has more than one SymbolicInt, then for some i, j, Ai = Aj. The particular sig will be counted twice in the second fact. Also note that the new Alloy syntax requires the second fact to be written using the plus function as the + operator is dedicated to set union operation.

```
#SymbolicInt = (#A1)·plus[(#A2)·plus[(#A3)·plus[ ···]]]
```

6.2.2.4 Alloy Analyzer usage

This section discusses a practical issue in analyzing a model that contains symbolic clauses instead of concrete integers. The key problem is to deal with redundant clauses that may exist in a symbolic instance because they are allowed by the chosen scope, although not explicitly enforced by the constraints, i.e., to separate redundant clauses from enforced clauses. Recall that the Alloy Analyzer finds valid instances of the given model for the given scope. Any instance with redundant clauses within given bounds is still valid. These redundant clauses are not bound to any particular condition on the symbolic variables and can take many possible values resulting in the Alloy Analyzer showing many instances that are only different in the values of redundant clauses. We present two approaches to address this problem.

Iterative deepening The first approach is to iteratively run the Alloy Analyzer on increasing scopes for `Clause` atoms until we find a solution. The predicates in `symbolic` module require certain `Clause` atom to exist. If the scope for `sig Clause` is smaller than the number of required clauses, then the Alloy Analyzer will declare that no solutions can be found. This separate bound on `sig Clause` can be given as:

```
run RepOk for 3 but 1 Clause
```

There are three considerations in this approach. The first is performance. Performance is an issue for large models where the bound on `Clause` has to be tested from zero to some larger bound. However, for most models, Alloy analysis is often performed for small sizes. Thus the repetitions required for testing different values is also expected to be small. Still, this incurs a performance overhead.

The second consideration is how to decide an upper bound on number of clauses. The user may use multiple clauses on each symbolic variable. We can enumerate to twice the number of symbolic variables as a safe bound and then inform the user that there may be instances with more clauses but none with fewer clauses. If the user knows that their model needs more clauses, then they can give a higher bound for the clauses to find such instances.

The third consideration is if we find a solution with *n* clauses, there may be solutions with more than *n* clauses. For example, the user can write a predicate like:

```
a·eq[b] || (a·eq[c] && c·eq[b])
```

Such an expression can result in one to three clauses. If Alloy Analyzer finds a solution with *n* clauses, there might be solutions with $n+1$ and $n+2$ clauses. Because of this, when we find a valid solution, we inform the user that there might be solutions with more clauses. Again, the user – with knowledge of the model – can force a higher bound on clauses or rewrite such predicates.

6.2.2.5 Skolemization

The second approach for handling the bound on `Clause` atoms uses *skolemization* in Alloy. According to Alloy's quick guide, "Often times, quantified formulas can be reduced to equivalent formulas without the use of quantifiers. This reduction is called *skolemization* and is based on the introduction of one or more skolem constants or functions that capture the constraint of the quantified formula in their values."

The important aspect of skolemization for our purpose is that skolemized atoms are identified explicitly in Alloy Analyzer's output. If we ensure that all generated clauses are skolemized we can start with a large bound for `Clause` atoms and easily identify redundant `Clause` atoms in the output.

Additionally, Alloy Analyzer's code can be modified to generate only skolemized atoms of one kind. This eliminates all issues related with bounds on the number of clauses. Only enforced clauses will be generated.

The only drawback to this scheme is that the user needs to ensure all predicates can be converted by skolemization. For example, the ordering check for sorted list in Section 6.2.1 does not produce skolemized results the way it is written. However the following equivalent predicate does:

```
some tail: l·header·*nextNode | no tail·nextNode
  && all n: l·header·*nextNode-tail | (n·data)·lt[n·nextNode·data]
```

Instead of an implication, we have to use universal and existential quantifiers. The new sorting check for linked list works with skolemization.

Skolemization translates existential quantifier based expressions. In the future, it should be investigated if the technique associated with skolemization – that renames an atom generated to satisfy a predicate – can be separately used for symbolic execution of Alloy. This would require changing the Alloy Analyzer implementation and only allowing `Clause` atoms that are generated to satisfy predicates in the `symbolic` module. Such `Clause` atoms would be generated regardless of how the predicate in `symbolic` module was invoked.

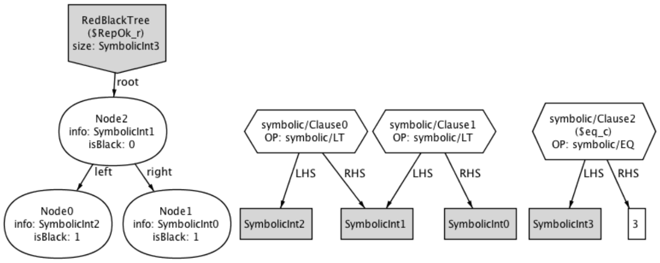
6.2.3 Case Studies

This section presents four small case studies that demonstrate that our technique enables novel forms of analysis of Alloy models using the Alloy Analyzer.

6.2.3.1 Red-Black Trees

Red-black trees [26] are binary search trees with one extra bit of information per node: its color, which can be either red or black. By restricting the way nodes are colored on a path from the root to a leaf, red-black trees ensure that the tree is balanced, i.e., guarantee that basic dynamic set operations on a red-black tree take O(lg n) time in the worst case.

Figure 6.3 Visualizing the constraints on data in a red-black tree with three nodes.



A binary search tree is a red-black tree if:

1. Every node is either red or black.
2. Every leaf (NIL) is black.
3. If a node is red, then both its children are black.
4. Every path from the root node to a descendant leaf contains the same number of black nodes.

All four of these red-black properties are expressible in Alloy [63]. Each node is modeled as:

```
sig Node {
  left: Node,
  right: Node,
  data: SymbolicInt,
  isBlack: Bool }
```

The core binary tree properties are:

```
pred isBinaryTree(r: RedBlackTree) {
  all n: r·root·*(left + right) {
    n !in n·^(left + right) -- no directed cycle
    lone n·~(left + right) -- at most one parent
    no n·left & n·right }} -- distinct children
```

We show how symbolic execution of Alloy formulas helps in generating and visualizing red-black tree instances. Using symbolic execution for `size` is similar to sorted linked list. We now show how to make `data` symbolic and write the binary search tree ordering constraints using predicates in the `symbolic` module.

```
pred isOrdered(r: RedBlackTree) {
  all n: r·root·*(left+ right) { -- ordering constraint
    some n·left ⇒ (n·left·info)·lt[n·info]
    some n·right ⇒ (n·info)·lt[n·right·info] }}
```

Next, we consider the `isBlack` relation. The constraints to validate color are:

```
pred isColorOk(r: RedBlackTree) {
  all e: root·*(left + right) | -- red nodes have black children
    e·isBlack = false && some e·left + e·right ⇒
      (e·left + e·right)·isBlack = true

  all e1, e2: root·*(left + right) | --all paths have same #blacks
    (no e1·left || no e1·right) && (no e2·left || no e2·right) ⇒
      #{ p: root·*(left+ right) |
        e1 in p·*(left+ right) && p·isBlack = true } =
      #{ p: root·*(left+ right) |
        e2 in p·*(left+ right) && p·isBlack = true }
}
```

We don't want `isBlack` to be symbolic because `isBlack` ensures that the generated trees are balanced. If we allow `isBlack` to be symbolic, the Alloy Analyzer will give instances with unbalanced trees combined with a set of unsolvable constraints for `isBlack`. To avoid such instances we keep `isBlack` concrete.

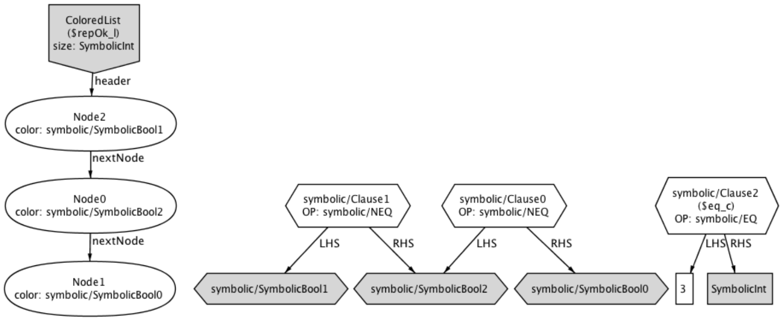
In Figure 6.3, an example of a red-black tree instance produced by symbolic execution of the above model is shown. The `root` node is red while both children are black. The constraints show that `data` in `left` node has to be less than data in `root` node which has to be less than data in the `right` node. Another constraint shows that `size` has to be three for this red-black tree.

6.2.3.2 Colored List

In this example, we consider a list where no two successive elements have the same color. This example presents a case where symbolic booleans are used.

The `Node sig` is defined as:

Figure 6.4 Visualizing the constraints on a list with alternating colors. Presents an example with symbolic booleans.



```
sig Node {
 nextNode: lone Node,
 color: SymbolicBool }
```

The check for alternate colors in the list can be written as

```
pred ColorsOk(l: ColoredList) {
  all n: l·header·*nextNode |
    some n·nextNode ⇒ (n·color)·neq[n·nextNode·color] }
```

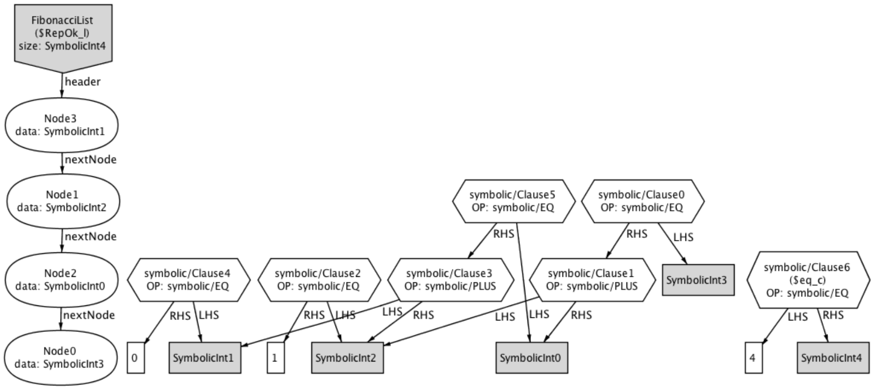
When this Alloy model is symbolically executed, one instance we get is shown in Figure 6.4. There are expressions that restrict the value of each boolean to be *not equal* to either its predecessor's data or its successor's data.

Such a models help in visualizing the structure of a model and understanding the relationships between various elements. Since each symbolic instance corresponds to a class of concrete instances, we are able to visualize more structures and build a better understanding of the model in much less time.

6.2.3.3 Fibonacci Series

This example presents how symbolic execution of Alloy models is able to allow non-trivial numeric operations and help avoid integer overflow. Because of Alloy's SAT-based analysis, the domain of integers used has to be kept small and integer overflow is a well-recognized issue. The Alloy 4.2 release candidate supports an option that disables generation of instances that have numeric overflow. Our approach provides an alternative solution since we build constraints on symbolic fields and do not require SAT to perform arithmetic.

Figure 6.5 Visualizing the constraints on data in a fibonacci sequence. Presents an example of non-trivial numeric constraints.



This example considers a fibonacci series stored in a linked list. The first two elements are required to contain zero and one. Anything after that contains the sum of last two elements. This can be modeled in Alloy as:

```
pred isFibonacci(l: SortedList) {
  some l·header ⇒ (l·header·data)·eq[0]
  some l·header·nextNode ⇒ (l·header·nextNode·data)·eq[1]
  all n: l·header·*nextNode |
    let p = n·nextNode, q = p·nextNode |
      some q ⇒ (n·data)·plus[p·data] &&
      (q·data)·eq[(n·data)·plus[p·data]] }
```

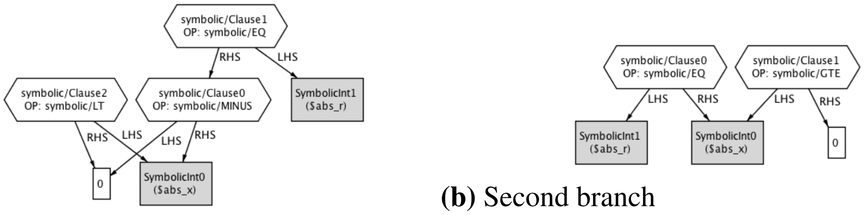
The first two constraints ensure that if the header and its next exist, they should be equal to 0 and 1 respectively. The third constraint works on all nodes (n) that have two more nodes (p and q) in front of them. It generates a plus clause between n and p and then generates an equality clause between the plus clause and q. This covers all restrictions on data in a fibonacci series.

Figure 6.5 shows an instance of the fibonacci list with four nodes. The conditions show that the third and fourth node have to contain the sum of the previous two, while the first two nodes can only contain 0 and 1. This shows the expressive power of symbolic execution for Alloy models and the way it shows a whole class of concrete inputs in a single visualization.

6.2.3.4 Traditional symbolic execution of imperative code

This section demonstrates an example of a small imperative function that is translated to Alloy and is symbolically executed using the Alloy Analyzer. This shows a non-conventional application of the Alloy Analyzer. Consider the abs function from Section 2.1 that returns the absolute value of its input.

Figure 6.6 Visualizing constraints on two paths within a small imperative function. Presents an example of visualizing traditional path conditions using Alloy.



(b) Second branch

(a) First branch

```
static int abs(int x) {
  int result;
  if (x < 0)
```

```
    result = 0 - x;
  else result = x;
  return result; }
```

This function can modeled in Alloy as:

```
pred abs(x: Int, result: Int) {
  x·lt[0] ⇒ 0·minus[x] && result·eq[0·minus[x]]
    else x·gte[0] && result·eq[x] }
```

The predicate takes `x` and `result` where `x` is the original input and `result` models the return value of this function. Symbolic execution of this function explores two paths with conditions `x<0` on one path and `x>=0` on the other path.

When we run this model using symbolic execution for Alloy models, we find both these paths in the output of Alloy Analyzer. The visualization of these paths is shown in Figure 6.6. Within the correct bounds and when redundant clauses are prevented, these are the only two results generated by the Alloy Analyzer.

This case study is one of the novel applications of symbolic execution in Alloy. It shows that Alloy can even provide a symbolic execution engine for traditional symbolic execution. It is yet to be seen how feasible Alloy would be in comparison with other symbolic execution engines for analysis of imperative programs.

Chapter 7

Related work

The idea of using constraints for representing test inputs has been used for at least three decades [21, 53, 65, 87] and implemented in EFFIGY [65], TESTGEN [66], and INKA [46] among other tools. However most of this work was to solve constraints on primitive data like integers and not structural constraints.

Goodenough and Gerhart [45] discuss the importance of specification based testing. Test case generation has been automated from specifications by many tools. Some examples are from Z specifications [32], UML statecharts [82], ADL specifications [18], and AsmL specifications [48]. However these specifications are also targeted to primitive types and not structurally complex inputs.

Constraints on complex structures require very different constraint solving techniques, which have only been explored more recently. Directions of research include using model checkers [41, 111], SAT solvers [101], symbolic execution [42, 91], and specialized solvers [10]. Section 6.1.1 discusses some tools that embody these techniques with examples.

One common problem faced while generating complex structures is isomorphism [93]. Two structures are defined to be isomorphic if they only differ in object identities. For example, if all elements in two nodes of a tree are swapped and all references to these nodes are swapped too, the resulting structure is identical to the original except that pointer values in some nodes would be different. Since, most programs are not concerned with the actual pointer *values* and only with *where* they are pointing, generating isomorphic structures is considered redundant and the algorithms attempt isomorph breaking procedures to reduce generated structures.

In the following sections, we describe further details of related work on symbolic execution, Korat, SAT solvers, and parallel algorithms in this domain.

7.1 Symbolic execution

Clarke [22] and King [65] pioneered traditional symbolic execution for imperative programs with primitive types. Much progress has been made on symbolic execution during the last decade. PREfix [13] is among the first systems to show the bug finding ability of symbolic execution on real code. Generalized symbolic execution [64] shows how to apply traditional symbolic execution to object-oriented code and uses *lazy initialization* to handle pointer aliasing. In contrast to generalized symbolic execution which solves integer constraints and produces *one* test case for each path in `repOk`, our work on multi-value Korat is centered around reducing the number of `repOk` executions to produce *all* test cases.

The main problem with symbolic execution is that for large or complex units, it is computationally infeasible to maintain and solve the constraints required for test generation. Larson and Austin [70] combined symbolic execution with concrete execution to overcome this limitation. Their approach was limited as they used symbolic execution to make the path constraint of a concrete execution and find other input values that can lead to errors along the same path.

Symbolic execution guided by concrete inputs has been a topic of extensive investigation during the last six years. DART [42] combines concrete and symbolic execution to collect the branch conditions along the execution path. DART negates the last branch condition to construct a new path condition that can drive the function to execute on another path. DART focuses only on path conditions involving integers.

To overcome the path explosion in large programs, SMART [40] introduced inter-procedural static analysis techniques to compute procedure summaries and reduce the paths to be explored by DART. SMART's procedure summaries bear resemblance to abstract symbolic tests but serve a very different purpose — summaries allow symbolic excution to avoid following method calls, whereas abstract symbolic tests are expanded into concrete tests as required during the second stage of staged symbolic execution.

CUTE [91] extends DART to handle constraints on references. EGT [15] and EXE [16] also use the negation of branch predicates and symbolic execution to generate test cases. They increase the precision of symbolic pointer analysis to handle pointer arithmetic and bit-level memory locations. Another symbolic execution tool CREST [12] was introduced for comparing various search strategies. This is the tool we are basing our one of our parallel techniques upon. KLEE [14] is the most recent tool from the EGT/EXE family. KLEE is open-sourced and has been used by a variety of users in academia and industry. KLEE works on LLVM byte code [1]. It works on unmodified programs written in C/C++ and has been shown to work for many off the shelf programs. Our work on ranged symbolic execution uses KLEE as an enabling technology.

Hybrid concolic testing [74] uses random search to periodically guide symbolic execution to increase code coverage. However, it can explore overlapping ranges when hopping from symbolic execution in one area of code to another, since no exploration boundaries are defined (other than time out). Ranged symbolic execution can in fact enable a novel form of hybrid concolic testing, which avoids overlapping ranges by hopping outside of the ranges already explored and not reentering them.

Directed incremental symbolic execution [86] leverages differences among program versions to optimize symbolic execution of *affected* paths that may exhibit modified behavior. The basic motivation is to avoid symbolically executing paths that have already been explored in a previous program version that was symbolically executed. A reachability analysis is used to identify affected locations, which guide the symbolic exploration. We believe ranged symbolic execution can provide an alternative technique for incremental symbolic execution where program edits are summarized as test pairs that are computed based on the edit locations and the pairs provide the ranges for symbolic execution.

All the above approaches require verifying the input pre-conditions and running the program under test together. Thus there is no concept of storing symbolic tests as part of test suites. The only tests that can be stored are concrete tests that are generated after analysis of the program under test.

Abstract subsumption checking [3] presents an approach for testing an under-approximation of the program using symbolic execution. It introduces abstractions for lists and arrays and checks if a symbolic state is subsumed by an earlier state under the abstractions. If so, the search is backtracked. This way symbolic execution can substantially reduce the number of explored paths for the program under-approximated using the abstractions.

Generational search [43] has suggested better search strategies for symbolic execution. They use the observation that any different clauses in the path condition can be negated and solved to visit different branches. Negating the first results in traversing the not-taken side of the first branch while negating the last results in traversing the not-taken side of the last branch. The former is a breadth-first strategy while the later is a depth-first strategy. They develop a heuristic to take the branches that will maximize code coverage. While their work is focused on providing a better search strategy, we focus on providing a parallel implementation and evaluating it.

Symbolic Execution has been applied outside the domain of imperative programs. Thums and Balser [108] use symbolic execution to verify temporal logic and statecharts. They consider every possible transition and maintain the symbolic state. Wang et al. [114] use symbolic execution to analyze behavioral requirements represented as Live Sequence Charts (LSC). LSC are executable specifications that allow the designer to work out aberrant scenarios. Symbolic execution allows them to group a number of concrete scenarios that only differ in the value of some variable. These are novel applications of symbolic execution, however, they translate the problem from some domain to a sequence of events with choices. This is essentially a sequential operation. To our knowledge symbolic execution has not yet been applied to declarative logic programs, which is what we do on our work on symbolic execution for Alloy.

7.2 Structural constraint solving using Korat

Korat implements a state-less search, in the spirit of VeriSoft [41], where backtracking is achieved through re-execution. The backtracking engine only stores the sequence of choices on the current path but not the entire program state at each choice point. The parallel search of PKorat applies to other analyses implemented using state-less search. To illustrate, consider symbolic execution implemented using the Java PathFinder (JPF) model checker [64], which uses JPF with state matching turned off. The delegation of work to slaves for symbolic execution would be identical to the PKorat technique, although a key challenge in scaling this technique for symbolic execution is to minimize communication among the processors—*path* *conditions* in symbolic execution can grow significantly and communicating them is likely to be expensive. Similarly, in principle, the parallel search of PKorat applies to combined symbolic/concrete execution [42, 91] and enables PKorat to perform parallel white-box testing. Our work on ranged symbolic execution uses these insights and provides a succint encoding of the state of a run of symbolic execution using a test input to enable a novel way to perform parallel symbolic execution.

Recent frameworks based on symbolic/concrete (aka concolic or dynamic symbolic) execution [16, 42, 91] that handle references/pointers are most closely related to Korat. A major difference is Korat's spirit of bounded *exhaustive* generation and backtracking based on last field accessed and not last branch taken. Generalized symbolic execution [64] follows Korat's spirit: lazy initialization of references has exactly the same effect as Korat's monitoring – both approaches consider the same candidates in the same order and generate the same structures. Practically, Korat is much faster since it is a specialized implementation–baseline Korat is an order of magnitude faster than a highly optimized version of lazy initialization on Java PathFinder [39]. In our experiments using CUTE [91] for structural constraint solving, Korat outperformed CUTE by two orders of magnitude. This is because of the overhead to keep symbolic state and Korat's specialized nature to backtrack on last accessed field. More recently, lazy initialization has also been implemented for equivalence checking of operations on complex structures in UC-KLEE [88].

Dedicated generators in Korat [75] are most closely related to our technique of multi-value comparisons in Korat. Dedicated generators exploit common input properties to efficiently generate inputs. Basic generators check if: (1) a value is in a set; (2) two values are equal; (3) two values are equal; (4) a values is less/greater than another value etc. There are high level generators that check if a pointer points to a tree or an acyclic graph etc. If the user takes the time to use the generator library, dedicated generators can be more efficient than our technique. However, for unmodified predicates, dedicated generators are not applicable, whereas we can still detect and apply our optimization. In one way, the `forwardFn` we introduced is a dedicated generator which is introduced automatically where applicable.

Glass-box testing [28] uses the method to be tested to prune Korat's generation. Thus it moves away from the pure black-box approach of Korat. Glass-box testing can be optimized using our technique. Efficient backtracking [35] optimizes Korat by using abstract undo operations that enable re-using partial `repOk` executions. However, it needs explicit support from the `repOk` writer in the form of using un-doable operations. STARC [34] uses the Korat algorithm to repair structures. Our approach of multi-value comparisons using Korat would make STARC efficient by forwarding over many invalid choices and thus reducing the number of `repOk` executions. Efficient backtracking improves the performance of exploring one candidate. Normally, for every candidate Korat runs `repOk` from the start. However most of the time, only the last accessed field is changed. This means that the initial part of the predicate will run unnecessarily. The initial part ends at the first access of a mutated field. Efficient backtracking uses this idea. Instead of Korat invoking `repOk` for every candidate, `repOk` invokes Korat for the next candidate and then undoes its operations up to the appropriate point. Unlike efficient backtracking, our approach of focused generation considers the other problem of what to generate and what not to generate.

UDITA [38] is a recently developed language that provides the ability to combine declarative and imperative predicates. It is based on JPF and delayed choice which is an extension of the lazy initialization algorithm.

7.2.1 SAT-based analysis — Alloy

SAT-based static analysis tools (such as JAlloy [57], CBMC [19]) can perform bounded exhaustive checking for heap-allocated data. However, they require a translation of the whole program *and* its specification to a SAT formula: for non-small programs the formulas choke the solvers. Korat requires solving only for input constraints, which are much simpler than the cumulative constraint that represents the correctness of the program under test. Static analysis tools that perform sound analysis of heap-allocated data (traditional shape analysis [89], verification conditions [36], separation logic [81]) often require more manual effort (in the form of loop invariants, additional predicates etc.) and have not been shown to scale to checking applications, which Korat readily handles.

The Alloy Analyzer uses the Kodkod tool [110], which provides the interface to SAT. The Alloy tool-set also includes JForge [30], which is a framework for analyzing a Java procedure against strong specifications within given bounds. It uses Kodkod for its analysis. JForge translates an imperative Java program to its declarative equivalent. We believe JForge can provide an enabling technology to transform our technique for symbolic execution of Alloy models to handle imperative programs.

TestEra [76] uses the Alloy tool-set for test input generation. To our knowledge, TestEra is the first tool to provide structurally complex input generation for systematic black-box testing. Whispec [92] builds on TestEra and solves method pre-conditions written in Alloy but uses a form of dynamic symbolic execution to guide concrete test generation for increasing code coverage.

7.3 Parallel program analysis

7.3.1 Parallel Korat

Parallel testing using Korat has previously been explored in the context of test generation as well as test execution [77]. The focus of this dissertation is a new technique for parallel test generation. PKorat can directly use the previous algorithms for parallel test execution. For test generation, PKorat significantly improves on the previous work, which considered two strategies to parallelize test generation. The first strategy, SEQ-OFF/ON, executes Korat sequentially once to determine an optimal partitioning of the input space such that in a subsequent execution of Korat for the same input space and predicate, each slave explores the same number of candidate inputs. PKorat differs from SEQ-OFF/ON by not requiring an initial sequential run and can in fact be used to optimize SEQ-OFF/ON. The second strategy PAR-OFF/ON uses randomization to fast-forward Korat search on one machine to “guess” equidistant candidate vectors. However, experimental results show little speed-up and low efficiency for PAR-OFF/ON. For example, for generating DAGs with 7 nodes, PAR-OFF provides 1.41X speed-up on average for 16 workers, and 8.08X speed-up on average for 1024 workers.

7.3.2 Parallel Symbolic Execution

Static partitioning [103] uses an initial shallow run of symbolic execution to minimize the communication overhead during parallel symbolic execution. The key idea is to create pre-conditions using conjunctions of clauses on path conditions encountered during the shallow run and to restrict symbolic execution by each worker to program paths that satisfy the pre-condition for that worker’s path exploration. However, the creation of pre-conditions results in different workers exploring overlapping ranges, which results in wasted effort.

On the other hand, our approach ParSym parallelizes symbolic execution by treating every path exploration as a unit of work and using a central server to distribute work between parallel workers. Our work on ranged symbolic execution uses dynamic load balancing, ensures workers have no overlap (other than on the paths that define the range boundaries), and keeps the communication low. Both of these parallel algorithms differ from static partitioning in that they ensure no two parallel workers analyze an overlapping range and thus cause wasted effort.

KleeNet [90] uses KLEE to find interaction bugs in distributed applications by running the distributed components under separate KLEE instances and coordinating them using a network model. KleeNet performs separate symbolic execution tasks of each component of the distributed application in parallel. However, it has no mechanism of parallelizing a single symbolic execution task.

7.3.3 Other Parallel Dynamic Analysis

Parallel search algorithms in general have long been studied [47, 58, 61]. Only recently, however, they have been used for searching state spaces in the area of model checking and program testing.

Parallel model checkers have been introduced. For example, Stern and Dill's parallel Murϕ [104] is an example of a parallel model checker. It keeps the set of visited states shared between parallel workers so that the same parts of the state space are not searched by multiple workers. Keeping this set synchronized between the workers results in expensive communication so the algorithm does not scale well.

A similar technique was used by Lerda and Visser [112] to parallelize the Java PathFinder model checker [72]. Parallel version of the SPIN model checker [51] was produced by Lerda and Sisto [71]. More work has been done in load balancing and reducing worker communication in these algorithms [59, 67, 84].

Parallel Randomized State Space Search for JPF by Dwyer et al. [33] takes a different approach with workers exploring randomly different parts of the state space. This often speeds up time to find first error with no worker communication. However when no errors are present, every worker has to explore every state. Our work on PKorat differs in that no two workers explore the same state.

7.3.4 Parallel Frameworks

There are general parallelization frameworks both on clusters and multicore machines. Cilk [9] is one the most popular framework for multicore or multiprocessor shared memory machines. It uses a work stealing algorithm that forms the basis of our distributed work stealing algorithm. Load balancing on distributed systems using work stealing [69] is also a well-studied topic. We use symbolic execution specific units of work and use a similar work stealing approach. PVM [107] was a popular framework for clusters but has been largely made obsolete by MPI (Message Passing Interface) which is a standard implemented on many platforms. However these frameworks provide the basis for implementing more specialized algorithms like in our case; we build our distributed work stealing on top of MPI.

Chapter 8

Conclusions

This dissertation introduced Pikse, a novel methodology for more effective and efficient checking of code conformance to specifications using parallel and incremental techniques, described a prototype implementation that embodies the methodology, and presented experiments that demonstrate its efficacy. Pikse has at its foundation a well-studied approach – *systematic constraint-driven analysis* – that has two common forms: (1) *constraint-based testing* – where logical constraints that define desired inputs and expected program behavior are used for test input generation and correctness checking, say to perform *black-box testing*; and (2) *symbolic execution* – where a systematic exploration of (bounded) program paths using *symbolic* input values is used to check properties of program behavior, say to perform *white-box testing*.

Our insight at the heart of Pikse was that for certain path-based analyses, (1) the state of a run of the analysis can be encoded compactly, which provides a basis for parallel techniques that have low communication overhead; and (2) iterations performed by the analysis have commonalities, which provides the basis for incremental techniques that re-use results of computations common to successive iterations. We embodied our insight into a suite of parallel and incremental techniques that enabled more effective and efficient constraint-driven analysis. We presented a series of experiments to evaluate our techniques. Experiemntal results showed Pikse enables significant speedups over previous state-of-the-art.

Bibliography

[1] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. “LLVA: A Low-level Virtual Instruction Set Architecture”. In: *Proc. 36th International Symposium on Microarchitecture (MICRO)*. 2003, pp. 205–216 (cit. on pp. 78, 174).

[2] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. “JPF-SE: a Symbolic Execution Extension to Java PathFinder”. In: *Proc. 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2007, pp. 134–138 (cit. on p. 25).

[3] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. “Symbolic Execution with Abstraction”. In: *International Journal Software Tools Technology Transfer* 11 (1 2009) (cit. on p. 175).

[4] Cyrille Artho, Doron Drusinksy, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Grigore Rosu, and Willem Visser. “Experiments with Test Case Generation and Runtime Analysis”. In: *Proc. 10th International Workshop on Abstract State Machines (ASM)*. 2003, pp. 87–107 (cit. on p. 132).

[5] Thomas Ball and Sriram K. Rajamani. “Automatically Validating Temporal Safety Properties of Interfaces”. In: *Proc. 8th International SPIN Workshop on Model Checking of Software*. 2001, pp. 103–122 (cit. on p. 131).

[6] Clark Barrett and Cesare Tinelli. “CVC3”. In: *Proc. 19th International Conference on Computer Aided Verification (CAV)*. 2007, pp. 298–302 (cit. on p. 58).

[7] Rudolf Bayer. “Symmetric Binary B-trees: Data Structure and Maintenance Algorithms”. In: *Acta informatica* 1.4 (Dec. 1972), pp. 290–306 (cit. on p. 131).

[8] Dirk Beyer, Adam J. Chlipala, and Rupak Majumdar. “Generating Tests from Counterexamples”. In: *Proc. 2004 International Conference on Software Engineering (ICSE)*. 2004, pp. 326–335 (cit. on p. 131).

[9] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. “Cilk: An Efficient Multithreaded Runtime System”. In: *Proc. 55th Conference on Principles and Practice of Parallel Programming (PPOPP)*. 1995, pp. 207–216 (cit. on p. 182).

[10] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. “Korat: Automated Testing based on Java Predicates”. In: *Proc. 2002 International Symposium on Software Testing and Analysis (ISSTA)*. 2002, pp. 123–133 (cit. on pp. 1, 3, 10, 14, 66, 71, 78, 86, 100–102, 108, 118, 123, 130, 139, 143, 172).

[11] Guillaume Brat, Doron Drusinsky, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Arnaud Venet, Willem Visser, and Rich Washington. “Experimental Evaluation of Verification and Validation Tools on Martian Rover Software”. In: *Formal Methods Systems Design* 25.2-3 (Sept. 2004), pp. 167–198 (cit. on p. 132).

[12] Jacob Burnim and Koushik Sen. “Heuristics for Scalable Dynamic Test Generation”. In: *Proc. 23rd International Conference on Automated Software Engineering (ASE)*. 2008, pp. 443–446 (cit. on pp. 10, 58, 66, 174).

[13] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. “A Static Analyzer for Finding Dynamic Programming Errors”. In: *Software Practice Experience* 30.7 (June 2000), pp. 775–802 (cit. on pp. 136, 173).

[14] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *Proc. 8th Symposium on Operating Systems Design and Implementation (OSDI)*. 2008, pp. 209–224 (cit. on pp. 10, 25, 118, 119, 121, 122, 174).

[15] Cristian Cadar and Dawson Engler. “Execution Generated Test Cases: How to make systems code crash itself”. In: *Proc. International SPIN Workshop on Model Checking of Software*. 2005, pp. 2–23 (cit. on p. 174).

[16] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. “EXE: Automatically Generating Inputs of Death”. In: *Proc. 13th Conference on Computer and Communications Security (CCS)*. 2006, pp. 322–335 (cit. on pp. 24, 111, 137, 174, 177).

[17] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. “Symbolic execution for software testing in practice: preliminary assessment”. In: *Proc. 33rd International Conference on Software Engineering (ICSE)*. 2011, pp. 1066–1071 (cit. on pp. 1, 12).

[18] Juei Chang and Debra J. Richardson. “Structural Specification-based Testing: Automated Support and Experimental Evaluation”. In: *Proc. 2nd joint meeting of the European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*. 1999, pp. 285–302 (cit. on p. 172).

[19] Edmund Clarke, Daniel Kroening, and Flavio Lerda. “A Tool for Checking ANSI-C Programs”. In: *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2004), pp. 168–176 (cit. on p. 179).

[20] Edmund M. Clarke. “The Birth of Model Checking”. In: *25 Years of Model Checking: History, Achievements, Perspectives*. 2008, pp. 1–26 (cit. on p. 131).

[21] Lori A. Clarke. “A System to Generate Test Data and Symbolically Execute Programs”. In: *IEEE Transactions on Software Engineering* 2.3 (May 1976), pp. 215–222 (cit. on p. 172).

[22] Lori A. Clarke. “Test Data Generation and Symbolic Execution of Programs as an aid to Program Validation.” PhD thesis. University of Colorado at Boulder, 1976 (cit. on pp. 1–3, 12, 25, 119, 173).

[23] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzé. “Using Symbolic Execution for Verifying Safety-critical Systems”. In: *Proc. 3rd joint meeting of the European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*. 2001, pp. 142–151 (cit. on p. 136).

[24] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. “The AETG System: An Approach to Testing Based on Combinatorial Design”. In: *IEEE Trans. Softw. Eng.* 23 (7 July 1997), pp. 437–444 (cit. on p. 1).

[25] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, and Hongjun Zheng. “Bandera: Extracting Finite-state Models from Java Source Code”. In: *Proc. 2000 International Conference on Software Engineering (ICSE)*. 2000, pp. 439–448 (cit. on p. 131).

[26] Thomas T. Cormen et al. *Introduction to Algorithms*. MIT Press, 1990. ISBN: 0-262-03141-8 (cit. on pp. 10, 165).

[27] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. “Automated Testing of Refactoring Engines”. In: *Proc. 6th joint meeting of the European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*. 2007, pp. 185–194 (cit. on pp. 10, 108).

[28] Paul T. Darga and Chandrasekhar Boyapati. “Efficient Software Model Checking of Data Structure Properties”. In: *Proc. 21st Annual Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 2006, pp. 363–382 (cit. on p. 178).

[29] Richard A. DeMillo and A. Jefferson Offutt. “Constraint-Based Automatic Test Data Generation”. In: *IEEE Trans. Softw. Eng.* 17 (9 Sept. 1991), pp. 900–910 (cit. on p. 1).

[30] G. Dennis and K Yessenov. *Forge website*. `http://sdg.csail.mit.edu/forge/` (cit. on p. 179).

[31] Hyunsook Do and Gregg Rothermel. “On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques”. In: *IEEE Trans. Softw. Eng.* 32 (9 2006) (cit. on p. 125).

[32] Michael R. Donat. “Automating Formal Specification-Based Testing”. In: *Proc. 7th International joint Conference on CAAP/FASE on Theory and Practice of Software Development (TAPSOFT)*. 1997, pp. 833–847 (cit. on p. 172).

[33] Matthew B. Dwyer, Sebastian Elbaum, Suzette Person, and Rahul Purandare. “Parallel Randomized State-Space Search”. In: *Proc. 2007 International Conference on Software Engineering (ICSE)*. 2007, pp. 3–12 (cit. on p. 182).

[34] Bassem Elkarablieh, Sarfraz Khurshid, Duy Vu, and Kathryn S. McKinley. “STARC: Static Analysis for Efficient Repair of Complex Data”. In: *Proc. 22nd Annual Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 2007, pp. 387–404 (cit. on p. 178).

[35] Bassem Elkarablieh, Darko Marinov, and Sarfraz Khurshid. “Efficient Solving of Structural Constraints”. In: *Proc. 2008 International Symposium on Software Testing and Analysis (ISSTA)*. 2008, pp. 39–50 (cit. on p. 178).

[36] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. “Extended Static Checking for Java”. In: *Proc. 2002 Conference on Programming Languages Design and Implementation (PLDI)*. 2002, pp. 234–245 (cit. on pp. 136, 179).

[37] Juan Pablo Galeotti et al. “Analysis of Invariants for Efficient Bounded Verification”. In: *Proc. International Symposium on Software Testing and Analysis (ISSTA)*. 2010 (cit. on p. 108).

[38] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. “Test generation through programming in UDITA”. In: *Proc. 2010 International Conference on Software Engineering (ICSE)*. 2010 (cit. on pp. 14, 108, 178).

[39] Milos Gligoric, Tihomir Gvero, Steven Lauterburg, Darko Marinov, and Sarfraz Khurshid. “Optimizing Generation of Object Graphs in Java PathFinder”. In: *Proc. 2nd International Conference on Software Testing Verification and Validation (ICST)*. 2009, pp. 51–60 (cit. on pp. 132, 177).

[40] Patrice Godefroid. “Compositional Dynamic Test Generation”. In: *Proc. 34th Symposium on Principles of Programming Languages (POPL)*. 2007, pp. 47–54 (cit. on pp. 24, 174).

[41] Patrice Godefroid. “Model Checking for Programming Languages using VeriSoft”. In: *Proc. 24th Symposium on Principles of Programming Languages (POPL)*. 1997, pp. 174–186 (cit. on pp. 172, 176).

[42] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: Directed Automated Random Testing”. In: *Proc. 2005 Conference on Programming Languages Design and Implementation (PLDI)*. 2005, pp. 213–223 (cit. on pp. 1, 51, 52, 57, 111, 119, 136, 137, 172, 173, 177).

[43] Patrice Godefroid, Michael Y. Levin, and David A Molnar. “Automated Whitebox Fuzz Testing”. In: *Proc. Network and Distributed System Security Symposium (NDSS)*. 2008 (cit. on pp. 24, 175).

[44] Eugene Goldberg and Yakov Novikov. “BerkMin: A Fast and Robust SAT-solver”. In: *Discrete Applied Math* 155.12 (June 2007), pp. 1549–1561 (cit. on p. 134).

[45] John B. Goodenough and Susan L. Gerhart. “Toward a Theory of Test Data Selection”. In: *Proc. International Conference on Reliable Software Technology*. 1975, pp. 493–510 (cit. on p. 172).

[46] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. “Automatic test data generation using constraint solving techniques”. In: *Proc. International Symposium on Software Testing and Analysis (ISSTA)*. 1998, pp. 53–62 (cit. on pp. 1, 172).

[47] Ananth Grama and Vipin Kumar. “State of the Art in Parallel Search Techniques for Discrete Optimization Problems”. In: *IEEE Transactions on Knowledge and Data Engineering* 11.1 (Jan. 1999), pp. 28–35 (cit. on p. 181).

[48] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. “Generating Finite State Machines from Abstract State Machines”. In: *Proc. 2002 International Symposium on Software Testing and Analysis (ISSTA)*. 2002, pp. 112–122 (cit. on p. 172).

[49] Leo J. Guibas and Robert Sedgewick. “A Dichromatic Framework for Balanced Trees”. In: *Proc. 19th Annual Symposium on Foundations of Computer Science (FOCS)*. 1978, pp. 8–21 (cit. on pp. 131, 135).

[50] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. “Software Verification with BLAST”. In: *Proc. 10th International SPIN Workshop on Model Checking of Software*. 2003, pp. 235–239 (cit. on p. 131).

[51] Gerard J. Holzmann. “The Model Checker SPIN”. In: *IEEE Transactions on Software Engineering* 23.5 (May 1997), pp. 279–295 (cit. on p. 181).

[52] W. E. Howden. “Symbolic Testing and the DISSECT Symbolic Evaluation System”. In: *IEEE Trans. Softw. Eng.* 3 (4 July 1977), pp. 266–278 (cit. on p. 1).

[53] Jung-chang Huang. “An Approach to Program Testing”. In: *ACM Computing Surveys* 7.3 (Sept. 1975), pp. 113–128 (cit. on pp. 1, 172).

[54] Daniel Jackson. “Alloy: A Lightweight Object Modelling Notation”. In: *ACM Transactions on Software Engineering and Methodology* 11.2 (Apr. 2002), pp. 256–290 (cit. on pp. 134, 135).

[55] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006 (cit. on pp. 1, 2, 21, 22, 118, 130, 154, 155).

[56] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. “ALCOA: The Alloy Constraint Analyzer”. In: *Proc. 2000 International Conference on Software Engineering (ICSE)*. 2000, pp. 730–733 (cit. on p. 134).

[57] Daniel Jackson and Mandana Vaziri. “Finding bugs with a constraint solver”. In: *Proc. 2000International Symposium on Software Testing and Analysis (ISSTA)*. 2000, pp. 14–25 (cit. on p. 179).

[58] Virendra K. Janakiram, Dharma P. Agrawal, and Ravi Mehrotra. “A Randomized Parallel Backtracking Algorithm”. In: *IEEE Transactions on Computers* 37.12 (Dec. 1988), pp. 1665–1676 (cit. on p. 181).

[59] Michael D. Jones and Jacob Sorber. “Parallel Search for LTL Violations”. In: *International Journal Software Tools Technology Transfer* 7.1 (Feb. 2005), pp. 31–42 (cit. on p. 182).

[60] Darko Marinov Junaid Haroon Siddiqui and Sarfraz Khurshid. “Lightweight data-flow analysis for execution driven constraint solving”. In: *Proc. 5^{th} International Conference on Software Testing Verification and Validation (ICST)*. 2012 (cit. on p. 71).

[61] Richard M. Karp and Yanjun Zhang. “Randomized Parallel Algorithms for Backtrack Search and Branch-and-bound Computation”. In: *Journal of the ACM* 40.3 (July 1993), pp. 765–789 (cit. on p. 181).

[62] Sarfraz Khurshid and Darko Marinov. “Checking Java Implementation of a Naming Architecture Using TestEra”. In: *Electronics Notes Theory Computer Science* 55.3 (2001) (cit. on p. 130).

[63] Sarfraz Khurshid and Darko Marinov. “TestEra: Specification-Based Testing of Java Programs using SAT”. In: *Automated Software Engineering Journal* 11.4 (Oct. 2004), pp. 403–434 (cit. on pp. 1, 134, 152, 153, 166).

[64] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. “Generalized Symbolic Execution for Model Checking and Testing”. In: *Proc. 9^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2003, pp. 553–568 (cit. on pp. 1, 5, 14, 108, 109, 112, 117, 131, 136, 173, 176, 177).

[65] James C. King. “Symbolic Execution and Program Testing”. In: *Communications ACM* 19.7 (July 1976), pp. 385–394 (cit. on pp. 1–3, 12, 13, 25, 119, 136, 172, 173).

[66] Bogdan Korel. “Automated test data generation for programs with procedures”. In: *Proc. International Symposium on Software Testing and Analysis (ISSTA)*. ISSTA ’96. 1996, pp. 209–215 (cit. on pp. 1, 172).

[67] Rahul Kumar and Eric G. Mercer. “Load Balancing Parallel Explicit State Model Checking”. In: *Electronics Notes Theory Computer Science* 128.3 (Apr. 2005), pp. 19–34 (cit. on p. 182).

[68] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing*. 2nd. Addison Wesley, 2003 (cit. on pp. 59, 95).

[69] Vipin Kumar, Ananth Y. Grama, and Nageshwara Rao Vempaty. “Scalable load balancing techniques for parallel computers”. In: *J. Parallel Distrib. Comput.* 22 (1 July 1994), pp. 60–79 (cit. on p. 182).

[70] Eric Larson and Todd Austin. “High Coverage Detection of Input-related Security Faults”. In: *Proc. 12th USENIX Security Symposium on*. 2003, p. 9 (cit. on pp. 136, 173).

[71] Flavio Lerda and Riccardo Sisto. “Distributed-Memory Model Checking with SPIN”. In: *Proc. 5th International SPIN Workshop on Model Checking of Software*. 1999, pp. 22–39 (cit. on p. 181).

[72] Flavio Lerda and Willem Visser. “Addressing Dynamic Issues of Program Model Checking”. In: *Proc. 8th International SPIN Workshop on Model Checking of Software*. 2001, pp. 80–102 (cit. on p. 181).

[73] Barbara Liskov and John Guttag. *Program Development in JAVA: Abstraction, Specification, and Object-oriented Design*. Addison-Wesley Longman Publishing, 2000 (cit. on p. 15).

[74] Rupak Majumdar and Koushik Sen. “Hybrid Concolic Testing”. In: *Proc. 29th International Conference on Software Engineering (ICSE)*. 2007, pp. 416–426 (cit. on p. 174).

[75] Darko Marinov. “Automatic Testing of Software with Structurally Complex Inputs”. PhD thesis. Massachusetts Institute of Technology, 2005 (cit. on p. 177).

[76] Darko Marinov and Sarfraz Khurshid. “TestEra: A Novel Framework for Automated Testing of Java Programs”. In: *Proc. 16th International Conference on Automated Software Engineering (ASE)*. 2001, p. 22 (cit. on pp. 14, 108, 130, 179).

[77] Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov. “Parallel Test Generation and Execution with Korat”. In: *Proc. 6th joint meeting of the European Software Engineering Conference and Symposium on Foundations of Software Engineering (ES-EC/FSE)*. 2007, pp. 135–144 (cit. on pp. 94, 98, 106, 153, 180).

[78] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. “Chaff: Engineering an Efficient SAT Solver”. In: *Proc. 38th Design Automation Conference on (DAC)*. 2001, pp. 530–535 (cit. on p. 134).

[79] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2008, pp. 337–340 (cit. on pp. 3, 12, 137).

[80] George C. Necula, Scott Mcpeak, S. P. Rahul, and Westley Weimer. “CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs”. In: *Proc. 11th International Conference on Compiler Construction (CC)*. 2002, pp. 213–228 (cit. on p. 58).

[81] Huu Hai Nguyen, Viktor Kuncak, and Wei-Ngan Chin. “Runtime checking for separation logic”. In: *Proc. 19th International Conference on Verification, model checking, and abstract interpretation*. VMCAI’08. 2008, pp. 203–217. ISBN: 3-540-78162-5, 978-3-540-78162-2 (cit. on p. 179).

[82] Jeff Offutt and Aynur Abdurazik. “Generating Tests from UML Specifications”. In: *Proc. 2nd International Conference on Unified Modeling Language*. 1999, pp. 416–429 (cit. on p. 172).

[83] J. Offutt et al. “An Experimental Mutation System for Java”. In: *SIGSOFT Softw. Eng. Notes* 29.5 (2004) (cit. on p. 125).

[84] Robert Palmer and Ganesh Gopalakrishnan. “A Distributed Partial Order Reduction Algorithm”. In: *Proc. 22nd International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*. 2002, p. 370 (cit. on p. 182).

[85] John Penix, Willem Visser, Eric Engstrom, Aaron Larson, and Nicholas Weininger. “Verification of Time Partitioning in the DEOS Scheduler Kernel”. In: *Proc. 2000 International Conference on Software Engineering (ICSE)*. 2000, pp. 488–497 (cit. on p. 132).

[86] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. “Directed Incremental Symbolic Execution”. In: *Proc. 2011 Conference on Programming Languages Design and Implementation (PLDI)*. 2011, pp. 504–515 (cit. on pp. 24, 175).

[87] Chittoor V. Ramamoorthy, Siu-Bun F. Ho, and W. T. Chen. “On the Automated Generation of Program Test Data”. In: *IEEE Transactions on Software Engineering* 2.4 (July 1976), pp. 293–300 (cit. on pp. 1, 172).

[88] David A. Ramos and Dawson R. Engler. “Practical, Low-Effort Equivalence Verification of Real Code”. In: *Proc. 23rd International Conference on Computer Aided Verification (CAV)*. 2011, pp. 669–685 (cit. on pp. 14, 177).

[89] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. “Parametric shape analysis via 3-valued logic”. In: *ACM Trans. Program. Lang. Syst.* 24.3 (May 2002), pp. 217–298. ISSN: 0164-0925 (cit. on p. 179).

[90] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. “KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment”. In: *Proc. 9th International Conference on Information Processing in Sensor Networks (ISPN 2010)*. 2010, pp. 186–196 (cit. on p. 181).

[91] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A Concolic Unit Testing Engine for C”. In: *Proc. 5th joint meeting of the European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*. 2005, pp. 263–272 (cit. on pp. 1, 51, 52, 57, 66, 108, 109, 111, 118, 119, 137, 172, 174, 177).

[92] Danhua Shao, Sarfraz Khurshid, and Dewayne E. Perry. “Whispec: White-box Testing of Libraries using Declarative Specifications”. In: *Proc. LCSD*. 2007 (cit. on p. 179).

[93] Ilya Shlyakhter. “Generating Effective Symmetry-breaking Predicates for Search Problems”. In: *Discrete Applied Math* 155.12 (June 2007), pp. 1539–1548 (cit. on p. 172).

[94] Junaid Haroon Siddiqui and Sarfraz Khurshid. “ParSym: Parallel Symbolic Execution”. In: *Proc. 2nd International Conference on Software Technology and Engineering (ICSTE)*. 2010, V1: 405–409 (cit. on pp. 24, 51).

[95] Junaid Haroon Siddiqui and Sarfraz Khurshid. “PKorat: Parallel Generation of Structurally Complex Test Inputs”. In: *Proc. 2nd International Conference on Software Testing Verification and Validation (ICST)*. 2009, pp. 250–259 (cit. on pp. 71, 93).

[96] Junaid Haroon Siddiqui and Sarfraz Khurshid. “Staged Symbolic Execution”. In: *Proc. 27th Symposium on Applied Computing (SAC): Software Verification and Testing Track (SVT)*. 2012 (cit. on pp. 108, 109).

[97] Junaid Haroon Siddiqui and Sarfraz Khurshid. “Symbolic Alloy”. In: *Proc. International Conference on Formal Engineering Methods (ICFEM)*. 2011 (cit. on pp. 129, 154).

[98] Junaid Haroon Siddiqui, Dark Marinov, and Sarfraz Khurshid. “Optimizing a Structural Constraint Solver for Efficient Software Checking”. In: *Proc. 24th International Conference on Automated Software Engineering (ASE)*. 2009 (cit. on pp. 71, 88).

[99] Junaid Haroon Siddiqui, Darko Marinov, and Sarfraz Khurshid. “An Empirical Study of Structural Constraint Solving Techniques”. In: *Proc. International Conference on Formal Engineering Methods (ICFEM)*. 2009, pp. 88–106 (cit. on p. 129).

[100] Marc Snir and Steve Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, 1998 (cit. on pp. 10, 103).

[101] Niklas Sörensson and Niklas Een. “An Extensible SAT-solver”. In: *Proc. 6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*. 2003, pp. 502–518 (cit. on pp. 3, 134, 172).

[102] J. Michael Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., 1989 (cit. on p. 134).

[103] Matt Staats and Corina Păsăreanu. “Parallel Symbolic Execution for Structural Test Generation”. In: *Proc. 19th International Symposium on Software Testing and Analysis (ISSTA)*. 2010, pp. 183–194 (cit. on pp. 24, 42, 180).

[104] Ulrich Stern and David L. Dill. “Parallelizing the Murphi Verifier”. In: *Proc. 9th International Conference on Computer Aided Verification*. 1997, pp. 256–278 (cit. on p. 181).

[105] Keith Stobie. “Model Based Testing in Practice at Microsoft”. In: *Electronic Notes in Theoretical Computer Science* 111 (2005), pp. 5–12 (cit. on p. 14).

[106] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. "Software Assurance by Bounded Exhaustive Testing". In: *Proc. 2004 International Symposium on Software Testing and Analysis (ISSTA)*. 2004, pp. 133–142 (cit. on pp. 14, 108, 130).

[107] Vaidy S. Sunderam. "PVM: A Framework for Parallel Distributed Computing". In: *Concurrency: Practice Experience* 2.4 (Dec. 1990), pp. 315–339 (cit. on p. 182).

[108] Andreas Thums and Michael Balser. "Interactive Verification of Statecharts". In: *Proc. INT 2004* (cit. on p. 176).

[109] Nikolai Tillmann and Jonathan De Halleux. "Pex: White box Test Generation for .NET". In: *Proc. TAP*. 2008 (cit. on pp. 111, 137).

[110] Emina Torlak and Daniel Jackson. "Kodkod: A Relational Model Finder". In: *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2007 (cit. on p. 179).

[111] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. "Model Checking Programs". In: *Proc. 15th International Conference on Automated Software Engineering (ASE)*. 2000, p. 3 (cit. on pp. 131, 172).

[112] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon ParkSEUNGJOON Park, and Flavio Lerda. "Model Checking Programs". In: *Automated Software Engineering Journal* 10.2 (Apr. 2003), pp. 203–232 (cit. on pp. 118, 181).

[113] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. "Test Input Generation with Java PathFinder". In: *Proc. 2004 International Symposium on Software Testing and Analysis (ISSTA)*. 2004, pp. 97–107 (cit. on pp. 130, 150).

[114] Tao Wang et al. "Symbolic Execution of Behavioral Requirements". In: *Pract. Aspects Decl. Lang.* 2004 (cit. on p. 176).

[115] Tao Xie, Darko Marinov, and David Notkin. "Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests". In: *Proc. 29th International Conference on Automated Software Engineering (ASE)*. 2004, pp. 196–205 (cit. on p. 130).

Vita

Junaid Haroon Siddiqui received the Bachelor of Science and the Master of Science degrees in Computer Science from National University of Computer and Emerging Sciences (FAST-NU), Lahore, Pakistan. He taught at the same university as visiting instructor for the next six years, while also working at a number of software development companies in Lahore, Pakistan. He applied to the University of Texas at Austin for enrollment in their computer engineering program. He was accepted and started graduate studies in August, 2007.

Permanent address: 1626-A W 6^{th} St
Austin, TX 78703

This dissertation was typeset with LaTeX[†] by the author.

[†] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.